

---

# EdgeDB Documentation

*Release 0.5.0*

**magicstack**

**May 19, 2023**



# CONTENTS

<b>1 Get Started</b>	<b>3</b>
1.1 Quickstart . . . . .	3
1.1.1 1. Installation . . . . .	3
1.1.2 2. Initialize a project . . . . .	4
1.1.3 3. Set up your schema . . . . .	5
1.1.4 4. Run a migration . . . . .	6
1.1.5 5. Write some queries . . . . .	7
1.1.6 Onwards and upwards . . . . .	10
1.2 The CLI . . . . .	10
1.2.1 Installation . . . . .	10
1.2.2 See help commands . . . . .	11
1.2.3 Upgrade the CLI . . . . .	12
1.3 Instances . . . . .	13
1.3.1 Creating databases . . . . .	13
1.3.2 Managing instances . . . . .	13
1.3.3 Listing instances . . . . .	13
1.3.4 Further reference . . . . .	14
1.4 Projects . . . . .	14
1.4.1 Initializing . . . . .	15
1.4.2 Connection . . . . .	15
1.4.3 Using remote instances . . . . .	16
1.4.4 Unlinking . . . . .	17
1.4.5 Upgrading . . . . .	17
1.4.6 See info . . . . .	17
1.5 Schema . . . . .	17
1.5.1 Scalar types . . . . .	17
1.5.2 Object types . . . . .	18
1.5.3 Properties . . . . .	18
1.5.3.1 Required vs optional . . . . .	18
1.5.3.2 Constraints . . . . .	19
1.5.3.3 Computed properties . . . . .	19
1.5.4 Links . . . . .	19
1.5.4.1 Computed links . . . . .	21
1.5.4.2 Backlinks . . . . .	21
1.5.5 Constraints . . . . .	22
1.5.6 Indexes . . . . .	23
1.5.7 Schema mixins . . . . .	23
1.5.8 Polymorphism . . . . .	24
1.6 Migrations . . . . .	25
1.6.1 1. Write an initial schema . . . . .	25

1.6.2	2. Edit your schema files . . . . .	26
1.6.3	3. Generate a migration . . . . .	27
1.6.4	4. Apply the migration . . . . .	27
1.6.5	Data migrations . . . . .	27
1.6.5.1	Further reading . . . . .	29
1.7	EdgeQL . . . . .	29
1.7.1	Scalar literals . . . . .	30
1.7.2	Functions and operators . . . . .	32
1.7.3	Insert an object . . . . .	33
1.7.4	Nested inserts . . . . .	34
1.7.5	Select objects . . . . .	34
1.7.6	Filtering, ordering, and pagination . . . . .	35
1.7.7	Query composition . . . . .	37
1.7.8	Computed properties . . . . .	38
1.7.9	Update objects . . . . .	39
1.7.10	Delete objects . . . . .	40
1.7.11	Query parameters . . . . .	40
1.7.12	Subqueries . . . . .	42
1.7.13	Polymorphic queries . . . . .	43
1.7.14	Grouping objects . . . . .	44
1.8	Client Libraries . . . . .	45
1.8.1	Available libraries . . . . .	45
1.8.2	Usage . . . . .	45
1.8.3	Connection . . . . .	49
1.8.3.1	Using projects . . . . .	49
1.8.3.2	Using <code>EDGEDB_DSN</code> . . . . .	49
1.8.3.3	Using multiple environment variables . . . . .	50
1.8.3.4	Other mechanisms . . . . .	51
1.8.3.5	Reference . . . . .	51
1.9	How to read the docs . . . . .	51
1.10	Tooling . . . . .	52
<b>2</b>	<b>Schema</b> . . . . .	<b>53</b>
2.1	Primitives . . . . .	53
2.1.1	Scalar types . . . . .	53
2.1.2	Enums . . . . .	54
2.1.3	Arrays . . . . .	54
2.1.4	Tuples . . . . .	55
2.1.5	Ranges . . . . .	55
2.1.6	Sequences . . . . .	56
2.2	Object Types . . . . .	56
2.2.1	IDs . . . . .	57
2.2.2	Abstract types . . . . .	57
2.2.3	Inheritance . . . . .	57
2.2.3.1	Multiple Inheritance . . . . .	58
2.3	Properties . . . . .	59
2.3.1	Required properties . . . . .	59
2.3.2	Property cardinality . . . . .	59
2.3.3	Default values . . . . .	60
2.3.4	Readonly properties . . . . .	61
2.3.5	Constraints . . . . .	61
2.3.6	Annotations . . . . .	62
2.3.7	Abstract properties . . . . .	63
2.3.8	Link properties . . . . .	63

2.4	Links . . . . .	64
2.4.1	Defining links . . . . .	64
2.4.2	Link cardinality . . . . .	64
2.4.3	Required links . . . . .	65
2.4.4	Exclusive constraints . . . . .	65
2.4.5	Modeling relations . . . . .	66
2.4.5.1	Many-to-one . . . . .	66
2.4.5.2	One-to-many . . . . .	67
2.4.5.3	One-to-one . . . . .	67
2.4.5.4	Many-to-many . . . . .	68
2.4.6	Default values . . . . .	68
2.4.7	Link properties . . . . .	69
2.4.8	Deletion policies . . . . .	70
2.4.8.1	Target deletion . . . . .	70
2.4.8.2	Source deletion . . . . .	71
2.4.9	Polymorphic links . . . . .	72
2.4.10	Abstract links . . . . .	73
2.5	Computeds . . . . .	73
2.5.1	Leading dot notation . . . . .	74
2.5.2	Type and cardinality inference . . . . .	74
2.5.3	Common use cases . . . . .	75
2.5.3.1	Filtering . . . . .	75
2.5.3.2	Backlinks . . . . .	75
2.5.3.3	Created Timestamp . . . . .	76
2.6	Indexes . . . . .	77
2.6.1	Index on a property . . . . .	77
2.6.2	Index on an expression . . . . .	77
2.6.3	Index on multiple properties . . . . .	78
2.6.4	Index on a link property . . . . .	78
2.6.5	Specify a Postgres index type . . . . .	78
2.6.6	Annotate an index . . . . .	79
2.7	Constraints . . . . .	79
2.7.1	Constraints on properties . . . . .	80
2.7.1.1	Custom constraints . . . . .	81
2.7.2	Constraints on object types . . . . .	81
2.7.2.1	Computed constraints . . . . .	82
2.7.2.2	Composite constraints . . . . .	82
2.7.2.3	Partial constraints . . . . .	83
2.7.3	Constraints on links . . . . .	83
2.7.4	Constraints on custom scalars . . . . .	84
2.8	Aliases . . . . .	84
2.9	Annotations . . . . .	86
2.9.1	Standard annotations . . . . .	86
2.9.2	User-defined annotations . . . . .	87
2.10	Globals . . . . .	87
2.10.1	Setting global variables . . . . .	87
2.10.1.1	Cardinality . . . . .	89
2.10.1.2	Computed globals . . . . .	89
2.10.1.3	Usage in schema . . . . .	90
2.11	Access Policies . . . . .	90
2.11.1	Defining a global . . . . .	91
2.11.2	Defining a policy . . . . .	93
2.11.3	Policy types . . . . .	94
2.11.4	Resolution order . . . . .	95

2.11.5	Custom error messages . . . . .	97
2.11.6	Disabling policies . . . . .	98
2.11.7	Examples . . . . .	98
2.11.7.1	Super constraints . . . . .	101
2.12	Functions . . . . .	102
2.12.1	User-defined Functions . . . . .	102
2.13	Triggers . . . . .	103
2.14	Mutation rewrites . . . . .	107
2.14.1	Available variables . . . . .	109
2.14.2	Mutation rewrite as cached computed . . . . .	111
2.15	Inheritance . . . . .	111
2.15.1	Object types . . . . .	112
2.15.1.1	Multiple Inheritance . . . . .	112
2.15.1.2	Overloading . . . . .	113
2.15.2	Properties . . . . .	113
2.15.3	Links . . . . .	114
2.15.4	Constraints . . . . .	114
2.15.5	Annotations . . . . .	115
2.16	Extensions . . . . .	115
2.17	Future Behavior . . . . .	115
2.18	vs SQL and ORMs . . . . .	116
2.18.1	Comparison to SQL . . . . .	116
2.18.2	Comparison to ORMs . . . . .	117
2.19	Introspection . . . . .	117
2.19.1	Object types . . . . .	118
2.19.2	Scalar types . . . . .	120
2.19.3	Collection types . . . . .	122
2.19.3.1	Array . . . . .	122
2.19.3.2	Tuple . . . . .	124
2.19.4	Functions . . . . .	125
2.19.5	Triggers . . . . .	127
2.19.6	Mutation rewrites . . . . .	129
2.19.7	Indexes . . . . .	132
2.19.8	Constraints . . . . .	133
2.19.9	Operators . . . . .	135
2.19.10	Casts . . . . .	137
2.20	SDL . . . . .	139
2.21	Migrations . . . . .	140
2.22	Terminology . . . . .	140
<b>3</b>	<b>EdgeQL</b>	<b>143</b>
3.1	Literals . . . . .	143
3.1.1	Strings . . . . .	143
3.1.2	Booleans . . . . .	145
3.1.3	Numbers . . . . .	145
3.1.4	UUID . . . . .	146
3.1.5	Enums . . . . .	146
3.1.6	Dates and times . . . . .	147
3.1.7	Durations . . . . .	147
3.1.7.1	Exact durations . . . . .	148
3.1.7.2	Relative durations . . . . .	148
3.1.7.3	Date durations . . . . .	148
3.1.8	Ranges . . . . .	149
3.1.9	Bytes . . . . .	150

3.1.10	Arrays . . . . .	150
3.1.11	Tuples . . . . .	150
3.1.11.1	Indexing tuples . . . . .	151
3.1.12	JSON . . . . .	151
3.2	Sets . . . . .	152
3.2.1	Everything is a set . . . . .	152
3.2.2	Constructing sets . . . . .	152
3.2.3	Literals are singletons . . . . .	153
3.2.4	Empty sets . . . . .	153
3.2.5	Set references . . . . .	154
3.2.6	Multisets . . . . .	154
3.2.7	Checking membership . . . . .	155
3.2.8	Merging sets . . . . .	155
3.2.9	Finding common members . . . . .	155
3.2.10	Removing common members . . . . .	155
3.2.11	Coalescing . . . . .	156
3.2.12	Inheritance . . . . .	156
3.2.13	Aggregate vs element-wise operations . . . . .	157
3.2.14	Conversion to/from arrays . . . . .	158
3.2.15	Reference . . . . .	158
3.3	Paths . . . . .	159
3.3.1	Backlinks . . . . .	160
3.3.2	Link properties . . . . .	161
3.3.3	Path roots . . . . .	161
3.4	Types . . . . .	161
3.4.1	Type expressions . . . . .	162
3.4.2	Type casting . . . . .	162
3.4.3	Type intersections . . . . .	163
3.4.4	Type checking . . . . .	163
3.4.5	The <code>typeof</code> operator . . . . .	163
3.4.6	Introspection . . . . .	164
3.5	Parameters . . . . .	164
3.5.1	Usage with clients . . . . .	164
3.5.1.1	REPL . . . . .	164
3.5.1.2	Python . . . . .	164
3.5.1.3	JavaScript . . . . .	165
3.5.1.4	Go . . . . .	165
3.5.2	Parameter types and JSON . . . . .	165
3.5.3	Optional parameters . . . . .	166
3.5.4	What can be parameterized? . . . . .	166
3.6	Select . . . . .	166
3.6.1	Selecting objects . . . . .	167
3.6.2	Shapes . . . . .	168
3.6.2.1	Nested shapes . . . . .	168
3.6.2.2	Splats . . . . .	169
3.6.3	Filtering . . . . .	173
3.6.3.1	Filtering by ID . . . . .	174
3.6.3.2	Nested filters . . . . .	174
3.6.4	Ordering . . . . .	175
3.6.5	Pagination . . . . .	176
3.6.6	Computed fields . . . . .	177
3.6.7	Backlinks . . . . .	177
3.6.8	Subqueries . . . . .	179
3.6.9	Polymorphic queries . . . . .	179

3.6.9.1	Polymorphic sets . . . . .	179
3.6.9.2	Polymorphic fields . . . . .	180
3.6.9.3	Filtering polymorphic links . . . . .	181
3.6.10	Free objects . . . . .	182
3.6.11	With block . . . . .	182
3.7	Insert . . . . .	183
3.7.1	Basic usage . . . . .	184
3.7.2	Inserting links . . . . .	184
3.7.3	Nested inserts . . . . .	185
3.7.4	With block . . . . .	186
3.7.5	Conflicts . . . . .	186
3.7.5.1	Upserts . . . . .	187
3.7.5.2	Suppressing failures . . . . .	187
3.7.6	Bulk inserts . . . . .	188
3.8	Update . . . . .	188
3.8.1	Syntax . . . . .	189
3.8.1.1	Updating links . . . . .	189
3.8.1.2	With blocks . . . . .	190
3.8.1.3	See also . . . . .	190
3.9	Delete . . . . .	190
3.9.1	Clauses . . . . .	190
3.9.2	Link deletion . . . . .	191
3.9.2.1	Cascading deletes . . . . .	191
3.9.3	Return value . . . . .	192
3.10	For . . . . .	192
3.10.1	Bulk inserts . . . . .	192
3.11	Group . . . . .	193
3.12	With . . . . .	197
3.12.1	Subqueries . . . . .	198
3.12.2	Query parameters . . . . .	198
3.12.3	Module selection . . . . .	198
3.13	Path resolution . . . . .	199
3.13.1	Scopes . . . . .	200
3.13.1.1	Clauses & Nesting . . . . .	201
3.14	Transactions . . . . .	201
3.14.1	Client libraries . . . . .	202
3.14.1.1	TypeScript/JS . . . . .	202
3.14.1.2	Python . . . . .	202
3.14.1.3	Golang . . . . .	202
3.15	Design goals . . . . .	203
3.16	Follow along . . . . .	203
<b>4</b>	<b>Guides</b>	<b>205</b>
4.1	Tutorials . . . . .	205
4.1.1	Next.js . . . . .	205
4.1.1.1	Updating the homepage . . . . .	206
4.1.1.2	Initializing EdgeDB . . . . .	207
4.1.1.3	Loading posts with an API route . . . . .	210
4.1.1.4	Generating the query builder . . . . .	211
4.1.1.5	Rendering blog posts . . . . .	213
4.1.1.6	Deploying to Vercel . . . . .	215
4.1.1.7	Wrapping up . . . . .	217
4.1.2	FastAPI . . . . .	217
4.1.2.1	Prerequisites . . . . .	218

4.1.2.1.1	Create a project directory . . . . .	218
4.1.2.1.2	Install the dependencies . . . . .	218
4.1.2.1.3	Initialize the database . . . . .	218
4.1.2.1.4	Connect to the database . . . . .	219
4.1.2.2	Schema design . . . . .	219
4.1.2.3	Run a migration . . . . .	220
4.1.2.4	Build the API endpoints . . . . .	221
4.1.2.4.1	Users API . . . . .	221
4.1.2.4.2	Events API . . . . .	228
4.1.2.4.3	Browse the endpoints using the native OpenAPI doc . . . . .	234
4.1.2.5	Wrapping up . . . . .	236
4.1.3	Flask . . . . .	236
4.1.3.1	Prerequisites . . . . .	236
4.1.3.1.1	Install the dependencies . . . . .	236
4.1.3.1.2	Initialize the database . . . . .	236
4.1.3.1.3	Connect to the database . . . . .	237
4.1.3.2	Schema design . . . . .	237
4.1.3.3	Build the API endpoints . . . . .	238
4.1.3.3.1	Fetch actors . . . . .	238
4.1.3.3.2	Create actor . . . . .	240
4.1.3.3.3	Update actor . . . . .	243
4.1.3.3.4	Delete actor . . . . .	245
4.1.3.3.5	Create movie . . . . .	246
4.1.3.3.6	Additional movie endpoints . . . . .	248
4.1.3.4	Conclusion . . . . .	249
4.1.4	Phoenix . . . . .	249
4.1.4.1	Prerequisites . . . . .	249
4.1.4.2	Schema design . . . . .	251
4.1.4.3	Ecto schemas . . . . .	253
4.1.4.4	User authentication via GitHub . . . . .	254
4.1.4.5	Running web server . . . . .	262
4.1.5	Strawberry . . . . .	262
4.1.5.1	Prerequisites . . . . .	263
4.1.5.1.1	Install the dependencies . . . . .	263
4.1.5.1.2	Initialize the database . . . . .	263
4.1.5.1.3	Connect to the database . . . . .	264
4.1.5.2	Schema design . . . . .	264
4.1.5.3	Build the GraphQL API . . . . .	265
4.1.5.3.1	Write the GraphQL schema . . . . .	265
4.1.5.3.2	Query actors . . . . .	266
4.1.5.3.3	Mutate actors . . . . .	270
4.1.5.3.4	Query movies . . . . .	273
4.1.5.3.5	Mutate movies . . . . .	274
4.1.5.4	Conclusion . . . . .	277
4.2	Deployment . . . . .	277
4.2.1	AWS . . . . .	278
4.2.1.1	Prerequisites . . . . .	278
4.2.1.2	Quick Install with CloudFormation . . . . .	278
4.2.1.2.1	CloudFormation Web Portal . . . . .	278
4.2.1.2.2	CloudFormation CLI . . . . .	279
4.2.1.3	Manual Install with CLI . . . . .	279
4.2.1.3.1	Create a VPC . . . . .	279
4.2.1.3.2	Create a Gateway . . . . .	280
4.2.1.3.3	Create a Public Network ACL . . . . .	280

4.2.1.3.4	Create a Private Network ACL . . . . .	281
4.2.1.3.5	Create a Public Subnet in Availability Zone “A” . . . . .	282
4.2.1.3.6	Create a Private Subnet in Availability Zone “A” . . . . .	283
4.2.1.3.7	Create a Public Subnet in Availability Zone “B” . . . . .	284
4.2.1.3.8	Create a Private Subnet in Availability Zone “B” . . . . .	285
4.2.1.3.9	Create an EC2 security group . . . . .	286
4.2.1.3.10	Create an RDS Security Group . . . . .	286
4.2.1.3.11	Create an RDS Cluster . . . . .	287
4.2.1.3.12	Create a Load Balancer . . . . .	288
4.2.1.3.13	Create an ECS Cluster . . . . .	289
4.2.1.3.14	Create a local link to the new EdgeDB instance . . . . .	292
4.2.1.4	Health Checks . . . . .	292
4.2.2	Azure . . . . .	292
4.2.2.1	Prerequisites . . . . .	293
4.2.2.2	Provision an EdgeDB instance . . . . .	293
4.2.2.3	Health Checks . . . . .	295
4.2.3	DigitalOcean . . . . .	295
4.2.3.1	One-click Deploy . . . . .	296
4.2.3.1.1	Prerequisites . . . . .	296
4.2.3.2	Deploy with Managed PostgreSQL . . . . .	297
4.2.3.2.1	Prerequisites . . . . .	297
4.2.3.2.2	Create a managed PostgreSQL instance . . . . .	297
4.2.3.2.3	Provision a droplet . . . . .	297
4.2.3.2.4	Health Checks . . . . .	298
4.2.4	Fly.io . . . . .	298
4.2.4.1	Prerequisites . . . . .	299
4.2.4.2	Provision a Fly.io app for EdgeDB . . . . .	299
4.2.4.3	Create a PostgreSQL cluster . . . . .	300
4.2.4.4	Start EdgeDB . . . . .	301
4.2.4.5	Persist the generated TLS certificate . . . . .	301
4.2.4.6	Connecting to the instance . . . . .	301
4.2.4.6.1	From a Fly.io app . . . . .	302
4.2.4.6.2	From external application . . . . .	302
4.2.4.6.3	From your local machine . . . . .	303
4.2.4.7	Health Checks . . . . .	303
4.2.5	Google Cloud . . . . .	304
4.2.5.1	Prerequisites . . . . .	304
4.2.5.2	Create a project . . . . .	304
4.2.5.3	Provision a Postgres instance . . . . .	304
4.2.5.4	Create a Kubernetes cluster . . . . .	305
4.2.5.5	Configure service account . . . . .	305
4.2.5.6	Deploy EdgeDB . . . . .	306
4.2.5.7	Persist TLS Certificate . . . . .	306
4.2.5.8	Expose EdgeDB . . . . .	306
4.2.5.9	Get your instance’s DSN . . . . .	307
4.2.5.9.1	In development . . . . .	307
4.2.5.9.2	In production . . . . .	308
4.2.5.10	Health Checks . . . . .	308
4.2.6	Heroku . . . . .	308
4.2.6.1	Prerequisites . . . . .	308
4.2.6.2	Setup . . . . .	308
4.2.6.3	Create a PostgreSQL Add-on . . . . .	309
4.2.6.4	Add the EdgeDB Buildpack . . . . .	309
4.2.6.5	Use <code>start-edgedb</code> in the Procfile . . . . .	309

4.2.6.6	Deploy the App . . . . .	309
4.2.6.7	Health Checks . . . . .	309
4.2.7	Docker . . . . .	309
4.2.7.1	When to use the edgedb/edgedb Docker image . . . . .	310
4.2.7.2	How to use this image . . . . .	310
4.2.7.3	Data Persistence . . . . .	310
4.2.7.4	Schema Migrations . . . . .	311
4.2.7.5	Docker Compose . . . . .	311
4.2.7.6	Configuration . . . . .	311
4.2.7.6.1	Initial configuration . . . . .	311
4.2.7.6.2	Runtime configuration . . . . .	313
4.2.7.7	Health Checks . . . . .	313
4.2.8	Bare Metal . . . . .	313
4.2.8.1	Install the EdgeDB Package . . . . .	313
4.2.8.1.1	Debian/Ubuntu LTS . . . . .	314
4.2.8.1.2	CentOS/RHEL 7/8 . . . . .	314
4.2.8.2	Enable a systemd unit . . . . .	314
4.2.8.3	Set environment variables . . . . .	315
4.2.8.4	Set a password . . . . .	315
4.2.8.5	Link the instance with the CLI . . . . .	315
4.2.8.6	Upgrading EdgeDB . . . . .	316
4.2.8.6.1	Debian/Ubuntu LTS . . . . .	316
4.2.8.6.2	CentOS/RHEL 7/8 . . . . .	316
4.2.8.7	Health Checks . . . . .	316
4.2.9	Health Checks . . . . .	316
4.2.9.1	Check Instance Aliveness . . . . .	316
4.2.9.2	Check Instance Readiness . . . . .	317
4.3	Migration Patterns . . . . .	317
4.3.1	Making a property required . . . . .	317
4.3.2	Adding backlinks . . . . .	320
4.3.3	Changing the type of a property . . . . .	322
4.3.4	Changing a property to a link . . . . .	325
4.3.5	Adding a required link . . . . .	331
4.4	Cheatsheets . . . . .	336
4.4.1	Selecting data . . . . .	337
4.4.2	Inserting data . . . . .	340
4.4.3	Updating data . . . . .	343
4.4.4	Deleting data . . . . .	346
4.4.5	Using link properties . . . . .	346
4.4.5.1	Declaration . . . . .	346
4.4.5.2	Constraints . . . . .	347
4.4.5.3	Indexes . . . . .	347
4.4.5.4	Inserting . . . . .	348
4.4.5.5	Updating . . . . .	348
4.4.5.6	Querying . . . . .	349
4.4.6	Working with booleans . . . . .	350
4.4.7	Object types . . . . .	353
4.4.8	Declaring functions . . . . .	357
4.4.9	Declaring aliases . . . . .	358
4.4.10	Declaring annotations . . . . .	359
4.4.11	Using the CLI . . . . .	360
4.4.12	Using the REPL . . . . .	361
4.4.12.1	Commands . . . . .	362
4.4.12.2	Sample usage . . . . .	363

4.4.13	Administering an instance . . . . .	364
4.5	Contributing . . . . .	365
4.5.1	Code . . . . .	365
4.5.1.1	Building Locally . . . . .	365
4.5.1.2	Running Tests . . . . .	367
4.5.1.3	Dev Server . . . . .	367
4.5.1.4	Test Databases . . . . .	367
4.5.2	Documentation . . . . .	367
4.5.2.1	Guidelines . . . . .	368
4.5.2.2	Style . . . . .	368
4.5.2.3	Where to Find It . . . . .	368
4.5.2.4	How to Build It . . . . .	369
4.5.2.4.1	edgedb/edgedb . . . . .	369
4.5.2.4.2	Full Documentation Build . . . . .	369
4.5.2.5	Sphinx and reStructuredText . . . . .	369
4.5.2.5.1	reStructuredText Basics . . . . .	369
4.5.2.5.2	Sphinx Basics . . . . .	373
4.5.2.6	Rendering Code . . . . .	374
4.5.2.6.1	Inline Code . . . . .	374
4.5.2.6.2	Code Blocks . . . . .	374
4.5.2.6.3	Code Tabs . . . . .	377
4.5.2.7	Documenting EdgeQL . . . . .	378
4.5.2.7.1	Functions . . . . .	378
4.5.2.7.2	Operators . . . . .	379
4.5.2.7.3	Statements . . . . .	379
4.5.2.7.4	Types . . . . .	380
4.5.2.7.5	Keywords . . . . .	380
4.5.2.8	Documenting the EdgeQL CLI . . . . .	381
4.5.2.9	Documentation Versioning . . . . .	381
4.5.2.9.1	New in Version . . . . .	381
4.5.2.9.2	Changed in Version . . . . .	383
4.5.2.10	Other Useful Tricks . . . . .	383
4.5.2.10.1	Temporarily Disabling Linting . . . . .	383
4.5.2.10.2	Embedding a YouTube Video . . . . .	384
4.5.2.10.3	Displaying Illustrations . . . . .	384
4.5.3	General Guidelines . . . . .	384
4.5.4	Thank You! . . . . .	385
5	<b>Standard Library</b> . . . . .	387
5.1	Generic . . . . .	387
5.2	Sets . . . . .	391
5.3	Types . . . . .	400
5.4	Math . . . . .	406
5.5	Strings . . . . .	408
5.5.1	Regular Expressions . . . . .	417
5.5.1.1	Option Flags . . . . .	417
5.5.2	Formatting . . . . .	417
5.5.2.1	Date and time formatting options . . . . .	417
5.5.2.2	Number formatting options . . . . .	419
5.6	Booleans . . . . .	420
5.7	Numbers . . . . .	424
5.7.1	Mathematical functions . . . . .	425
5.7.2	Bitwise functions . . . . .	425
5.7.3	String parsing . . . . .	425

5.8	JSON . . . . .	435
5.8.1	Constructing JSON Values . . . . .	435
5.9	UUIDs . . . . .	442
5.10	Enums . . . . .	443
5.11	Dates and Times . . . . .	443
5.12	Arrays . . . . .	463
5.12.1	Constructing arrays . . . . .	463
5.12.2	Empty arrays . . . . .	463
5.12.3	Reference . . . . .	464
5.13	Tuples . . . . .	467
5.13.1	Constructing tuples . . . . .	467
5.13.2	Accessing elements . . . . .	467
5.13.3	Nesting tuples . . . . .	468
5.13.4	Type syntax . . . . .	468
5.14	Ranges . . . . .	469
5.14.1	Constructing ranges . . . . .	470
5.14.2	JSON representation . . . . .	471
5.14.3	Functions and operators . . . . .	472
5.14.4	Reference . . . . .	472
5.15	Bytes . . . . .	478
5.16	Sequences . . . . .	479
5.17	Base Objects . . . . .	481
5.18	Abstract Types . . . . .	482
5.18.1	Abstract Numeric Types . . . . .	483
5.18.2	Abstract Range Types . . . . .	483
5.19	Constraints . . . . .	484
5.20	System . . . . .	487
5.21	Config . . . . .	488
5.21.1	Configuration Parameters . . . . .	489
5.21.1.1	Connection settings . . . . .	489
5.21.1.2	Resource usage . . . . .	489
5.21.1.3	Query planning . . . . .	489
5.21.1.4	Query behavior . . . . .	489
5.21.1.5	Client connections . . . . .	490
5.22	Deprecated . . . . .	491
5.23	Scalar Types . . . . .	493
5.24	Collection Types . . . . .	494
5.25	Range Types . . . . .	494
5.26	Object Types . . . . .	494
5.27	Types and Sets . . . . .	494
5.28	Utilities . . . . .	494
<b>6</b>	<b>Client Libraries</b> . . . . .	<b>495</b>
6.1	Connection . . . . .	495
6.2	JavaScript . . . . .	496
6.3	Python . . . . .	496
6.4	Go . . . . .	496
6.5	Rust . . . . .	496
6.6	Dart . . . . .	496
6.7	.NET . . . . .	497
6.8	EdgeQL over HTTP . . . . .	497
6.8.1	Protocol . . . . .	497
6.8.1.1	GET request . . . . .	498
6.8.1.2	POST request . . . . .	498

6.8.1.3	Response . . . . .	498
6.8.2	Health Checks . . . . .	498
6.8.2.1	Aliveness . . . . .	499
6.8.2.2	Readiness . . . . .	499
6.9	GraphQL . . . . .	499
6.9.1	Basics . . . . .	499
6.9.1.1	Queries . . . . .	499
6.9.1.1.1	Filtering . . . . .	500
6.9.1.1.2	Ordering . . . . .	502
6.9.1.1.3	Paginating . . . . .	503
6.9.1.1.4	Variables . . . . .	506
6.9.2	Mutations . . . . .	506
6.9.2.1	Delete . . . . .	506
6.9.2.2	Insert . . . . .	508
6.9.2.3	Update . . . . .	509
6.9.3	Introspection . . . . .	510
6.9.4	Cheatsheet . . . . .	512
6.9.5	Setting up the extension . . . . .	515
6.9.6	Connection . . . . .	516
6.9.7	The protocol . . . . .	516
6.9.7.1	POST request (recommended) . . . . .	517
6.9.7.2	GET request . . . . .	517
6.9.7.3	Response format . . . . .	517
6.9.8	Known limitations . . . . .	518
<b>7</b>	<b>CLI</b> . . . . .	<b>519</b>
7.1	Connection flags . . . . .	520
7.1.1	Connection flags . . . . .	521
7.2	Network usage . . . . .	522
7.2.1	Version Check . . . . .	522
7.2.2	Disabling Version Check . . . . .	522
7.2.3	edgedb server and edgedb self upgrade . . . . .	523
7.3	edgedb . . . . .	523
7.3.1	Description . . . . .	523
7.3.2	Options . . . . .	523
7.3.3	Backslash Commands . . . . .	524
7.4	edgedb dump . . . . .	526
7.4.1	Description . . . . .	526
7.4.2	Options . . . . .	527
7.5	edgedb restore . . . . .	527
7.5.1	Description . . . . .	527
7.5.2	Options . . . . .	527
7.6	edgedb configure . . . . .	527
7.6.1	Description . . . . .	528
7.6.2	Actions . . . . .	528
7.6.3	Options . . . . .	528
7.7	edgedb watch . . . . .	528
7.8	edgedb migration . . . . .	529
7.8.1	edgedb migration create . . . . .	529
7.8.1.1	Options . . . . .	529
7.8.2	edgedb migration apply . . . . .	530
7.8.2.1	Options . . . . .	530
7.8.3	edgedb migration log . . . . .	530
7.8.3.1	Options . . . . .	530

7.8.4	edgedb migration status . . . . .	531
7.8.4.1	Options . . . . .	531
7.8.5	edgedb migration edit . . . . .	531
7.8.5.1	Options . . . . .	531
7.8.6	edgedb migration upgrade-check . . . . .	531
7.8.6.1	Options . . . . .	532
7.8.7	Setup . . . . .	532
7.9	edgedb migrate . . . . .	532
7.10	edgedb database create . . . . .	532
7.10.1	Description . . . . .	533
7.10.2	Options . . . . .	533
7.11	edgedb describe . . . . .	533
7.11.1	edgedb describe object . . . . .	533
7.11.1.1	Description . . . . .	533
7.11.1.2	Options . . . . .	533
7.11.2	edgedb describe schema . . . . .	533
7.11.2.1	Description . . . . .	534
7.11.2.2	Options . . . . .	534
7.12	edgedb list . . . . .	534
7.12.1	Description . . . . .	534
7.12.2	Types . . . . .	534
7.12.3	Options . . . . .	534
7.13	edgedb query . . . . .	535
7.13.1	Description . . . . .	535
7.13.2	Options . . . . .	535
7.14	edgedb analyze . . . . .	535
7.14.1	Options . . . . .	535
7.15	edgedb ui . . . . .	536
7.15.1	Description . . . . .	536
7.15.2	Options . . . . .	536
7.16	edgedb info . . . . .	536
7.16.1	Paths . . . . .	536
7.16.1.1	Options . . . . .	537
7.17	edgedb project . . . . .	537
7.17.1	edgedb project init . . . . .	537
7.17.1.1	Description . . . . .	537
7.17.1.1.1	EdgeDB Cloud . . . . .	537
7.17.1.2	Options . . . . .	537
7.17.2	edgedb project unlink . . . . .	538
7.17.2.1	Description . . . . .	538
7.17.2.2	Options . . . . .	538
7.17.3	edgedb project info . . . . .	538
7.17.3.1	Description . . . . .	539
7.17.3.2	Options . . . . .	539
7.17.4	edgedb project upgrade . . . . .	539
7.17.4.1	Description . . . . .	539
7.17.4.2	Options . . . . .	539
7.18	edgedb instance . . . . .	540
7.18.1	edgedb instance create . . . . .	540
7.18.1.1	Description . . . . .	540
7.18.1.1.1	EdgeDB Cloud . . . . .	540
7.18.1.2	Options . . . . .	540
7.18.2	edgedb instance link . . . . .	541
7.18.2.1	Description . . . . .	541

7.18.2.2 Options . . . . .	541
7.18.3 edgedb instance unlink . . . . .	541
7.18.3.1 Description . . . . .	541
7.18.3.2 Options . . . . .	542
7.18.4 edgedb instance list . . . . .	542
7.18.4.1 Description . . . . .	542
7.18.4.2 Options . . . . .	542
7.18.5 edgedb instance logs . . . . .	542
7.18.5.1 Description . . . . .	542
7.18.5.2 Options . . . . .	542
7.18.6 edgedb instance status . . . . .	542
7.18.6.1 Description . . . . .	543
7.18.6.2 Options . . . . .	543
7.18.7 edgedb instance start . . . . .	543
7.18.7.1 Description . . . . .	543
7.18.7.2 Options . . . . .	543
7.18.8 edgedb instance stop . . . . .	543
7.18.8.1 Description . . . . .	543
7.18.8.2 Options . . . . .	544
7.18.9 edgedb instance restart . . . . .	544
7.18.9.1 Description . . . . .	544
7.18.9.2 Options . . . . .	544
7.18.10 edgedb instance destroy . . . . .	544
7.18.10.1 Description . . . . .	544
7.18.10.2 Options . . . . .	544
7.18.11 edgedb instance revert . . . . .	544
7.18.11.1 Description . . . . .	545
7.18.11.2 Options . . . . .	545
7.18.12 edgedb instance reset-password . . . . .	545
7.18.12.1 Description . . . . .	545
7.18.12.2 Options . . . . .	545
7.18.13 edgedb instance upgrade . . . . .	545
7.18.13.1 Description . . . . .	546
7.18.13.2 Options . . . . .	546
7.19 edgedb server . . . . .	546
7.19.1 edgedb server info . . . . .	546
7.19.1.1 Description . . . . .	547
7.19.1.2 Options . . . . .	547
7.19.2 edgedb server install . . . . .	547
7.19.2.1 Description . . . . .	547
7.19.2.2 Options . . . . .	547
7.19.3 edgedb server list-versions . . . . .	547
7.19.3.1 Description . . . . .	547
7.19.3.2 Options . . . . .	548
7.19.4 edgedb server uninstall . . . . .	548
7.19.4.1 Description . . . . .	548
7.19.4.2 Options . . . . .	548
7.20 edgedb cloud . . . . .	548
7.20.1 edgedb cloud login . . . . .	548
7.20.2 edgedb cloud logout . . . . .	549
7.20.2.1 Options . . . . .	549
7.20.3 edgedb cloud secretkey . . . . .	549
7.20.3.1 edgedb cloud secretkey create . . . . .	549
7.20.3.1.1 Options . . . . .	549

7.20.3.2	edgedb cloud secretkey list . . . . .	550
7.20.3.2.1	Options . . . . .	550
7.20.3.3	edgedb cloud secretkey revoke . . . . .	550
7.20.3.3.1	Options . . . . .	550
7.20.4	Usage . . . . .	551
7.21	edgedb cli upgrade . . . . .	551
7.21.1	Description . . . . .	551
7.21.2	Options . . . . .	551
<b>8</b>	<b>Reference</b>	<b>553</b>
8.1	EdgeQL . . . . .	553
8.1.1	Lexical structure . . . . .	554
8.1.1.1	Identifiers . . . . .	554
8.1.1.2	Names and keywords . . . . .	555
8.1.1.3	Constants . . . . .	556
8.1.1.3.1	Strings . . . . .	556
8.1.1.3.2	Bytes . . . . .	558
8.1.1.3.3	Integers . . . . .	558
8.1.1.3.4	Real Numbers . . . . .	559
8.1.1.4	Punctuation . . . . .	560
8.1.1.5	Comments . . . . .	560
8.1.1.6	Operators . . . . .	560
8.1.2	Evaluation algorithm . . . . .	561
8.1.3	Shapes . . . . .	562
8.1.3.1	Shaping Query Results . . . . .	562
8.1.3.2	General Shaping Rules . . . . .	566
8.1.3.3	Losing Shapes . . . . .	566
8.1.4	Paths . . . . .	568
8.1.5	Casts . . . . .	568
8.1.5.1	Explicit Casts . . . . .	568
8.1.5.2	Assignment Casts . . . . .	569
8.1.5.3	Implicit Casts . . . . .	569
8.1.5.4	Casting Table . . . . .	570
8.1.6	Function calls . . . . .	570
8.1.7	Cardinality . . . . .	571
8.1.7.1	Terminology . . . . .	571
8.1.7.2	Functions and operators . . . . .	571
8.1.7.2.1	Aggregate operations . . . . .	572
8.1.7.2.2	Element-wise operations . . . . .	572
8.1.7.2.3	Cartesian products . . . . .	572
8.1.7.2.4	Per-input cardinality . . . . .	573
8.1.7.2.5	Type qualifiers . . . . .	573
8.1.7.2.6	Cardinality computation . . . . .	573
8.1.8	Select . . . . .	573
8.1.8.1	Description . . . . .	574
8.1.8.2	Filter . . . . .	576
8.1.8.3	Clause signatures . . . . .	577
8.1.9	Insert . . . . .	577
8.1.9.1	Description . . . . .	577
8.1.9.2	Outputs . . . . .	578
8.1.9.3	Examples . . . . .	578
8.1.10	Update . . . . .	580
8.1.10.1	Output . . . . .	581
8.1.10.2	Examples . . . . .	581

8.1.11	Delete . . . . .	582
8.1.11.1	Output . . . . .	583
8.1.11.2	Examples . . . . .	583
8.1.12	For . . . . .	583
8.1.12.1	Usage of <code>for</code> statement . . . . .	584
8.1.13	Group . . . . .	586
8.1.13.1	Output . . . . .	587
8.1.13.2	Examples . . . . .	588
8.1.14	With block . . . . .	592
8.1.14.1	Specifying a module . . . . .	592
8.1.14.2	Local Expression Aliases . . . . .	593
8.1.14.3	Detached . . . . .	594
8.1.15	Start transaction . . . . .	595
8.1.15.1	Description . . . . .	595
8.1.15.2	Parameters . . . . .	595
8.1.15.3	Examples . . . . .	595
8.1.16	Commit . . . . .	596
8.1.16.1	Example . . . . .	596
8.1.16.2	Description . . . . .	596
8.1.17	Rollback . . . . .	596
8.1.17.1	Example . . . . .	597
8.1.17.2	Description . . . . .	597
8.1.18	Declare savepoint . . . . .	597
8.1.18.1	Description . . . . .	597
8.1.18.2	Example . . . . .	597
8.1.19	Release savepoint . . . . .	598
8.1.19.1	Description . . . . .	598
8.1.19.2	Example . . . . .	598
8.1.20	Rollback to savepoint . . . . .	599
8.1.20.1	Description . . . . .	599
8.1.20.2	Example . . . . .	599
8.1.21	Set . . . . .	599
8.1.21.1	Description . . . . .	600
8.1.21.2	Variations . . . . .	600
8.1.21.3	Examples . . . . .	601
8.1.22	Reset . . . . .	601
8.1.22.1	Description . . . . .	601
8.1.22.2	Variations . . . . .	601
8.1.22.3	Examples . . . . .	602
8.1.23	Describe . . . . .	602
8.1.23.1	Description . . . . .	603
8.1.23.2	Examples . . . . .	604
8.2	SDL . . . . .	606
8.2.1	Modules . . . . .	608
8.2.1.1	Example . . . . .	608
8.2.1.2	Syntax . . . . .	609
8.2.1.3	Description . . . . .	609
8.2.2	Object Types . . . . .	611
8.2.2.1	Example . . . . .	611
8.2.2.2	Syntax . . . . .	612
8.2.2.3	Description . . . . .	612
8.2.3	Scalar Types . . . . .	613
8.2.3.1	Example . . . . .	613
8.2.3.2	Syntax . . . . .	613

8.2.3.3	Description	613
8.2.4	Links	614
8.2.4.1	Examples	614
8.2.4.1.1	Overloading	614
8.2.4.2	Syntax	615
8.2.4.3	Description	617
8.2.5	Properties	618
8.2.5.1	Examples	618
8.2.5.2	Syntax	619
8.2.5.3	Description	620
8.2.6	Expression Aliases	621
8.2.6.1	Example	621
8.2.6.2	Syntax	621
8.2.6.3	Description	622
8.2.7	Indexes	622
8.2.7.1	Example	622
8.2.7.2	Syntax	623
8.2.7.3	Description	623
8.2.8	Constraints	623
8.2.8.1	Examples	623
8.2.8.2	Syntax	624
8.2.8.3	Description	624
8.2.9	Annotations	625
8.2.9.1	Examples	625
8.2.9.2	Syntax	626
8.2.9.3	Description	626
8.2.10	Globals	627
8.2.10.1	Examples	627
8.2.10.2	Syntax	627
8.2.10.3	Description	628
8.2.11	Access Policies	629
8.2.11.1	Examples	629
8.2.11.2	Syntax	630
8.2.11.3	Description	630
8.2.12	Functions	631
8.2.12.1	Example	632
8.2.12.2	Syntax	632
8.2.12.3	Description	633
8.2.13	Triggers	634
8.2.13.1	Example	634
8.2.13.2	Syntax	634
8.2.13.3	Description	635
8.2.14	Mutation rewrites	635
8.2.14.1	Example	635
8.2.14.2	Syntax	635
8.2.14.3	Description	636
8.2.15	Extensions	636
8.2.15.1	Syntax	636
8.2.15.2	Description	636
8.2.15.3	Examples	636
8.2.16	Future Behavior	636
8.2.16.1	Syntax	637
8.2.16.2	Description	637
8.2.16.3	Examples	637

8.3	DDL . . . . .	637
8.3.1	Modules . . . . .	637
8.3.1.1	Create module . . . . .	637
8.3.1.1.1	Description . . . . .	637
8.3.1.1.2	Parameters . . . . .	638
8.3.1.1.3	Examples . . . . .	638
8.3.1.2	Drop module . . . . .	638
8.3.1.2.1	Description . . . . .	638
8.3.1.2.2	Examples . . . . .	638
8.3.2	Object Types . . . . .	638
8.3.2.1	Create type . . . . .	639
8.3.2.1.1	Description . . . . .	639
8.3.2.1.2	Parameters . . . . .	639
8.3.2.1.3	Examples . . . . .	640
8.3.2.2	Alter type . . . . .	640
8.3.2.2.1	Description . . . . .	640
8.3.2.2.2	Parameters . . . . .	641
8.3.2.2.3	Examples . . . . .	642
8.3.2.3	Drop type . . . . .	642
8.3.2.3.1	Description . . . . .	642
8.3.2.3.2	Examples . . . . .	642
8.3.3	Scalar Types . . . . .	642
8.3.3.1	Create scalar type . . . . .	643
8.3.3.1.1	Description . . . . .	643
8.3.3.1.2	Examples . . . . .	643
8.3.3.2	Alter scalar type . . . . .	644
8.3.3.2.1	Description . . . . .	644
8.3.3.2.2	Examples . . . . .	645
8.3.3.3	Drop scalar type . . . . .	645
8.3.3.3.1	Description . . . . .	645
8.3.3.3.2	Parameters . . . . .	645
8.3.3.3.3	Example . . . . .	645
8.3.4	Links . . . . .	645
8.3.4.1	Create link . . . . .	646
8.3.4.1.1	Description . . . . .	647
8.3.4.1.2	Parameters . . . . .	647
8.3.4.1.3	Examples . . . . .	647
8.3.4.2	Alter link . . . . .	648
8.3.4.2.1	Description . . . . .	649
8.3.4.2.2	Parameters . . . . .	649
8.3.4.2.3	Examples . . . . .	650
8.3.4.3	Drop link . . . . .	651
8.3.4.3.1	Description . . . . .	651
8.3.4.3.2	Examples . . . . .	651
8.3.5	Properties . . . . .	651
8.3.5.1	Create property . . . . .	652
8.3.5.1.1	Description . . . . .	652
8.3.5.1.2	Parameters . . . . .	653
8.3.5.1.3	Examples . . . . .	653
8.3.5.2	Alter property . . . . .	653
8.3.5.2.1	Description . . . . .	654
8.3.5.2.2	Parameters . . . . .	654
8.3.5.2.3	Examples . . . . .	656
8.3.5.3	Drop property . . . . .	656

8.3.5.3.1	Description . . . . .	657
8.3.5.3.2	Example . . . . .	657
8.3.6	Aliases . . . . .	657
8.3.6.1	Create alias . . . . .	657
8.3.6.1.1	Description . . . . .	658
8.3.6.1.2	Parameters . . . . .	658
8.3.6.1.3	Example . . . . .	658
8.3.6.2	Drop alias . . . . .	658
8.3.6.2.1	Description . . . . .	658
8.3.6.2.2	Parameters . . . . .	659
8.3.6.2.3	Example . . . . .	659
8.3.7	Indexes . . . . .	659
8.3.7.1	Create index . . . . .	659
8.3.7.1.1	Description . . . . .	659
8.3.7.1.2	Parameters . . . . .	659
8.3.7.1.3	Example . . . . .	660
8.3.7.2	Alter index . . . . .	660
8.3.7.2.1	Description . . . . .	660
8.3.7.2.2	Parameters . . . . .	660
8.3.7.2.3	Example . . . . .	661
8.3.7.3	Drop index . . . . .	661
8.3.7.3.1	Description . . . . .	661
8.3.7.3.2	Example . . . . .	661
8.3.8	Constraints . . . . .	661
8.3.8.1	Create abstract constraint . . . . .	662
8.3.8.1.1	Description . . . . .	662
8.3.8.1.2	Parameters . . . . .	662
8.3.8.1.3	Example . . . . .	663
8.3.8.2	Alter abstract constraint . . . . .	663
8.3.8.2.1	Description . . . . .	663
8.3.8.2.2	Parameters . . . . .	663
8.3.8.2.3	Example . . . . .	664
8.3.8.3	Drop abstract constraint . . . . .	664
8.3.8.3.1	Description . . . . .	664
8.3.8.3.2	Parameters . . . . .	664
8.3.8.3.3	Example . . . . .	664
8.3.8.4	Create constraint . . . . .	665
8.3.8.4.1	Description . . . . .	665
8.3.8.4.2	Parameters . . . . .	665
8.3.8.4.3	Example . . . . .	666
8.3.8.5	Alter constraint . . . . .	666
8.3.8.5.1	Description . . . . .	667
8.3.8.5.2	Parameters . . . . .	667
8.3.8.5.3	Example . . . . .	667
8.3.8.6	Drop constraint . . . . .	667
8.3.8.6.1	Description . . . . .	668
8.3.8.6.2	Parameters . . . . .	668
8.3.8.6.3	Example . . . . .	668
8.3.9	Annotations . . . . .	668
8.3.9.1	Create abstract annotation . . . . .	668
8.3.9.1.1	Description . . . . .	669
8.3.9.1.2	Example . . . . .	669
8.3.9.2	Alter abstract annotation . . . . .	669
8.3.9.2.1	Description . . . . .	670

8.3.9.2.2	Parameters . . . . .	670
8.3.9.2.3	Examples . . . . .	670
8.3.9.3	Drop abstract annotation . . . . .	670
8.3.9.3.1	Description . . . . .	670
8.3.9.3.2	Example . . . . .	670
8.3.9.4	Create annotation . . . . .	671
8.3.9.4.1	Description . . . . .	671
8.3.9.4.2	Example . . . . .	671
8.3.9.5	Alter annotation . . . . .	671
8.3.9.5.1	Description . . . . .	671
8.3.9.5.2	Example . . . . .	671
8.3.9.6	Drop annotation . . . . .	672
8.3.9.6.1	Description . . . . .	672
8.3.9.6.2	Example . . . . .	672
8.3.10	Globals . . . . .	672
8.3.10.1	Create global . . . . .	672
8.3.10.1.1	Description . . . . .	673
8.3.10.1.2	Parameters . . . . .	673
8.3.10.1.3	Examples . . . . .	673
8.3.10.2	Alter global . . . . .	673
8.3.10.2.1	Description . . . . .	674
8.3.10.2.2	Parameters . . . . .	674
8.3.10.2.3	Examples . . . . .	675
8.3.10.3	Drop global . . . . .	675
8.3.10.3.1	Description . . . . .	675
8.3.10.3.2	Example . . . . .	675
8.3.11	Access Policies . . . . .	676
8.3.11.1	Create access policy . . . . .	676
8.3.11.1.1	Description . . . . .	677
8.3.11.1.2	Parameters . . . . .	677
8.3.11.2	Alter access policy . . . . .	678
8.3.11.2.1	Description . . . . .	679
8.3.11.2.2	Parameters . . . . .	679
8.3.11.3	Drop access policy . . . . .	679
8.3.11.3.1	Description . . . . .	679
8.3.12	Functions . . . . .	680
8.3.12.1	Create function . . . . .	680
8.3.12.1.1	Description . . . . .	681
8.3.12.1.2	Parameters . . . . .	681
8.3.12.1.3	Examples . . . . .	681
8.3.12.2	Alter function . . . . .	682
8.3.12.2.1	Description . . . . .	682
8.3.12.2.2	Subcommands . . . . .	682
8.3.12.2.3	Example . . . . .	683
8.3.12.3	Drop function . . . . .	683
8.3.12.3.1	Description . . . . .	683
8.3.12.3.2	Parameters . . . . .	683
8.3.12.3.3	Example . . . . .	684
8.3.13	Triggers . . . . .	684
8.3.13.1	Create trigger . . . . .	684
8.3.13.1.1	Description . . . . .	684
8.3.13.1.2	Parameters . . . . .	684
8.3.13.1.3	Example . . . . .	685
8.3.13.2	Drop trigger . . . . .	685

8.3.13.2.1	Description	685
8.3.13.2.2	Parameters	685
8.3.13.2.3	Example	685
8.3.14	Mutation Rewrites	686
8.3.14.1	Create rewrite	686
8.3.14.1.1	Description	686
8.3.14.1.2	Parameters	686
8.3.14.1.3	Examples	686
8.3.14.2	Drop rewrite	687
8.3.14.2.1	Description	687
8.3.14.2.2	Parameters	687
8.3.14.2.3	Example	687
8.3.15	Extensions	688
8.3.15.1	Create extension	688
8.3.15.1.1	Description	688
8.3.15.1.2	Examples	688
8.3.15.2	drop extension	688
8.3.15.2.1	Description	688
8.3.15.2.2	Examples	688
8.3.16	Future Behavior	689
8.3.16.1	Create future	689
8.3.16.1.1	Description	689
8.3.16.1.2	Examples	689
8.3.16.2	drop future	689
8.3.16.2.1	Description	689
8.3.16.2.2	Examples	689
8.3.17	Migrations	690
8.3.17.1	Start migration	690
8.3.17.1.1	Parameters	690
8.3.17.1.2	Description	690
8.3.17.1.3	Examples	691
8.3.17.2	create migration	691
8.3.17.2.1	Parameters	691
8.3.17.2.2	Description	691
8.3.17.2.3	Examples	691
8.3.17.3	Abort migration	692
8.3.17.3.1	Description	692
8.3.17.3.2	Examples	692
8.3.17.4	Populate migration	692
8.3.17.4.1	Description	692
8.3.17.4.2	Examples	693
8.3.17.5	Describe current migration	693
8.3.17.5.1	Description	693
8.3.17.6	Commit migration	694
8.3.17.6.1	Description	694
8.3.17.6.2	Example	695
8.3.17.7	Reset schema to initial	695
8.3.17.8	Migration Rewrites	695
8.3.17.8.1	Start migration rewrite	695
8.3.17.8.2	Declare savepoint	695
8.3.17.8.3	Release savepoint	696
8.3.17.8.4	Rollback to savepoint	696
8.3.17.8.5	Rollback	696
8.3.17.8.6	Commit migration rewrite	696

8.3.18	Comparison to SDL . . . . .	697
8.4	Connection parameters . . . . .	697
8.4.1	Specifying an instance . . . . .	697
8.4.2	Priority levels . . . . .	699
8.4.3	Granular parameters . . . . .	700
8.4.3.1	Override behavior . . . . .	701
8.4.3.2	Overriding across priority levels . . . . .	701
8.5	Environment Variables . . . . .	701
8.5.1	Variants . . . . .	702
8.5.2	Supported variables . . . . .	702
8.5.2.1	EDGEDB_SERVER_BOOTSTRAP_COMMAND . . . . .	702
8.5.2.2	EDGEDB_SERVER_DEFAULT_AUTH_METHOD . . . . .	702
8.5.2.3	EDGEDB_SERVER_TLS_CERT_MODE . . . . .	702
8.5.2.4	EDGEDB_SERVER_TLS_CERT_FILE/EDGEDB_SERVER_TLS_KEY_FILE . . . . .	703
8.5.2.5	EDGEDB_SERVER_SECURITY . . . . .	703
8.5.2.6	EDGEDB_SERVER_PORT . . . . .	703
8.5.2.7	EDGEDB_SERVER_BIND_ADDRESS . . . . .	703
8.5.2.8	EDGEDB_SERVER_DATADIR . . . . .	703
8.5.2.9	EDGEDB_SERVER_BACKEND_DSN . . . . .	703
8.5.2.10	EDGEDB_SERVER_RUNSTATE_DIR . . . . .	703
8.5.2.11	EDGEDB_SERVER_ADMIN_UI . . . . .	704
8.6	Create a project . . . . .	704
8.6.1	FAQ . . . . .	704
8.6.1.1	How does this help me? . . . . .	704
8.6.1.2	What do you mean <i>link</i> ? . . . . .	705
8.6.1.3	How does this work in production? . . . . .	705
8.6.1.4	What's the <code>edgedb.toml</code> file? . . . . .	705
8.6.1.5	How do I use <code>edgedb project</code> for existing codebases? . . . . .	705
8.6.1.6	How does this make projects more portable? . . . . .	706
8.6.1.7	How do I unlink a project? . . . . .	706
8.6.1.8	How do I use <code>edgedb project</code> with a non-local instance? . . . . .	706
8.7	DSN specification . . . . .	707
8.7.1	Query parameters . . . . .	707
8.8	Dump file format . . . . .	708
8.8.1	General Structure . . . . .	708
8.8.2	General Dump Block . . . . .	708
8.8.3	Header Block . . . . .	709
8.8.4	Data Block . . . . .	710
8.9	Backend high-availability . . . . .	710
8.9.1	API-based HA . . . . .	711
8.9.2	Adaptive HA . . . . .	711
8.10	Server configuration . . . . .	712
8.10.1	Configuring the server . . . . .	712
8.10.1.1	EdgeQL . . . . .	712
8.10.1.2	CLI . . . . .	712
8.10.2	Available settings . . . . .	713
8.10.2.1	Connection settings . . . . .	713
8.10.2.2	Resource usage . . . . .	713
8.10.2.3	Query planning . . . . .	713
8.10.2.4	Query behavior . . . . .	713
8.10.2.5	Client connections . . . . .	714
8.11	HTTP API . . . . .	714
8.11.1	Health Checks . . . . .	714
8.11.1.1	Aliveness . . . . .	714

8.11.1.2	Readiness . . . . .	715
8.11.2	Observability . . . . .	715
8.11.2.1	Processes . . . . .	715
8.11.2.2	Backend connections and performance . . . . .	715
8.11.2.3	Client connections . . . . .	715
8.11.2.4	Query compilation . . . . .	716
8.11.2.5	Errors . . . . .	716
8.11.3	Querying . . . . .	716
8.11.3.1	Making a query request . . . . .	716
8.11.3.2	Response . . . . .	717
8.12	SQL support . . . . .	717
8.12.1	Connecting . . . . .	717
8.12.2	Querying . . . . .	718
8.12.3	Tested SQL tools . . . . .	720
8.13	Binary protocol . . . . .	720
8.13.1	Messages . . . . .	721
8.13.1.1	ErrorResponse . . . . .	721
8.13.1.2	LogMessage . . . . .	723
8.13.1.3	ReadyForCommand . . . . .	723
8.13.1.4	RestoreReady . . . . .	724
8.13.1.5	CommandComplete . . . . .	724
8.13.1.6	Dump . . . . .	725
8.13.1.7	CommandDataDescription . . . . .	725
8.13.1.8	StateDataDescription . . . . .	726
8.13.1.9	Sync . . . . .	727
8.13.1.10	Restore . . . . .	727
8.13.1.11	RestoreBlock . . . . .	728
8.13.1.12	RestoreEof . . . . .	728
8.13.1.13	Execute . . . . .	728
8.13.1.14	Parse . . . . .	730
8.13.1.15	Data . . . . .	732
8.13.1.16	Dump Header . . . . .	732
8.13.1.17	Dump Block . . . . .	733
8.13.1.18	ServerKeyData . . . . .	734
8.13.1.19	ParameterStatus . . . . .	734
8.13.1.20	ClientHandshake . . . . .	735
8.13.1.21	ServerHandshake . . . . .	736
8.13.1.22	AuthenticationOK . . . . .	737
8.13.1.23	AuthenticationSASL . . . . .	737
8.13.1.24	AuthenticationSASLContinue . . . . .	738
8.13.1.25	AuthenticationSASLFinal . . . . .	739
8.13.1.26	AuthenticationSASLInitialResponse . . . . .	739
8.13.1.27	AuthenticationSASLResponse . . . . .	740
8.13.1.28	Terminate . . . . .	740
8.13.2	Errors . . . . .	740
8.13.2.1	Errors inheritance . . . . .	740
8.13.2.2	Error codes . . . . .	741
8.13.3	Type descriptors . . . . .	741
8.13.3.1	Set Descriptor . . . . .	741
8.13.3.2	Object Shape Descriptor . . . . .	741
8.13.3.3	Base Scalar Type Descriptor . . . . .	742
8.13.3.4	Scalar Type Descriptor . . . . .	743
8.13.3.5	Tuple Type Descriptor . . . . .	743
8.13.3.6	Named Tuple Type Descriptor . . . . .	744

8.13.3.7	Array Type Descriptor . . . . .	744
8.13.3.8	Enumeration Type Descriptor . . . . .	745
8.13.3.9	Input Shape Descriptor . . . . .	745
8.13.3.10	Range Type Descriptor . . . . .	745
8.13.3.11	Scalar Type Name Annotation . . . . .	746
8.13.3.12	Type Annotation Descriptor . . . . .	746
8.13.4	Data wire formats . . . . .	746
8.13.4.1	Sets and array<> . . . . .	747
8.13.4.2	tuple<>, namedtuple<>, and object<> . . . . .	748
8.13.4.3	Sparse Objects . . . . .	749
8.13.4.4	Ranges . . . . .	749
8.13.4.5	std::uuid . . . . .	750
8.13.4.6	std::str . . . . .	750
8.13.4.7	std::bytes . . . . .	750
8.13.4.8	std::int16 . . . . .	750
8.13.4.9	std::int32 . . . . .	750
8.13.4.10	std::int64 . . . . .	751
8.13.4.11	std::float32 . . . . .	751
8.13.4.12	std::float64 . . . . .	751
8.13.4.13	std::decimal . . . . .	751
8.13.4.14	std::bool . . . . .	752
8.13.4.15	std::datetime . . . . .	752
8.13.4.16	cal::local_datetime . . . . .	753
8.13.4.17	cal::local_date . . . . .	753
8.13.4.18	cal::local_time . . . . .	753
8.13.4.19	std::duration . . . . .	753
8.13.4.20	cal::relative_duration . . . . .	754
8.13.4.21	cal::date_duration . . . . .	754
8.13.4.22	std::json . . . . .	755
8.13.4.23	std::bigint . . . . .	755
8.13.4.24	cfg::memory . . . . .	756
8.13.5	Conventions and data Types . . . . .	756
8.13.6	Message Format . . . . .	757
8.13.7	Errors . . . . .	758
8.13.8	Logs . . . . .	758
8.13.9	Message Flow . . . . .	758
8.13.9.1	Connection Phase . . . . .	758
8.13.9.2	Authentication . . . . .	758
8.13.9.3	Command Phase . . . . .	759
8.13.9.4	Dump Database Flow . . . . .	760
8.13.9.5	Restore Database Flow . . . . .	760
8.13.10	Termination . . . . .	760
8.14	Client Libraries . . . . .	761
8.14.1	Date/Time Handling . . . . .	761
8.14.1.1	Precision . . . . .	762
8.15	Administration . . . . .	762
8.15.1	Configure . . . . .	763
8.15.1.1	Description . . . . .	763
8.15.1.2	Parameters . . . . .	763
8.15.1.3	Examples . . . . .	763
8.15.2	Database . . . . .	764
8.15.2.1	Create database . . . . .	764
8.15.2.1.1	Description . . . . .	764
8.15.2.1.2	Examples . . . . .	764

8.15.2.2	Drop database . . . . .	764
8.15.2.2.1	Description . . . . .	765
8.15.2.2.2	Examples . . . . .	765
8.15.3	Role . . . . .	765
8.15.3.1	Create role . . . . .	765
8.15.3.1.1	Description . . . . .	765
8.15.3.1.2	Examples . . . . .	766
8.15.3.2	Alter role . . . . .	766
8.15.3.2.1	Description . . . . .	766
8.15.3.2.2	Examples . . . . .	767
8.15.3.3	Drop role . . . . .	767
8.15.3.3.1	Description . . . . .	767
8.15.3.3.2	Examples . . . . .	767
<b>9</b>	<b>Changelog</b>	<b>769</b>
9.1	v1.0 . . . . .	769
9.1.1	1.4 . . . . .	769
9.1.2	1.3 . . . . .	770
9.1.3	1.2 . . . . .	770
9.1.4	1.1 . . . . .	771
9.1.5	Pre-releases . . . . .	772
9.2	v2.0 . . . . .	772
9.2.1	Upgrading . . . . .	772
9.2.1.1	Client libraries . . . . .	773
9.2.2	New features . . . . .	773
9.2.2.1	Integrated admin UI . . . . .	773
9.2.2.2	Analytical queries with GROUP . . . . .	774
9.2.2.3	Global variables . . . . .	775
9.2.2.4	Object-level security . . . . .	776
9.2.2.5	Range types . . . . .	776
9.2.2.6	The <code>cal::date_duration</code> type . . . . .	777
9.2.2.7	Source deletion policies . . . . .	777
9.2.3	Additional changes . . . . .	777
9.2.3.1	EdgeQL . . . . .	777
9.2.3.2	Server . . . . .	778
9.2.3.3	Bug fixes . . . . .	778
9.2.3.4	Protocol overhaul . . . . .	779
9.2.4	2.1 . . . . .	779
9.2.5	2.2 . . . . .	779
9.2.6	2.3 . . . . .	780
9.2.7	2.4 . . . . .	781
9.2.8	2.5 . . . . .	781
9.2.9	2.6 . . . . .	782
9.2.9.1	Nonrecursive access policies and future behaviors . . . . .	782
9.2.9.2	Other changes . . . . .	782
9.2.10	2.7 . . . . .	782
9.2.11	2.8 . . . . .	783
9.2.12	2.9 . . . . .	783
9.2.13	2.10 . . . . .	784
9.2.14	2.11 . . . . .	785
9.2.15	2.12 . . . . .	785
9.2.16	2.13 . . . . .	785
9.2.17	2.14 . . . . .	785
9.2.17.1	Schema repair on upgrades . . . . .	785

9.2.17.2	Other changes . . . . .	786
9.2.18	2.15 . . . . .	786
9.3	v3.0 (dev) . . . . .	786
9.3.1	Upgrading . . . . .	787
9.3.1.1	Client libraries . . . . .	788
9.3.2	New features . . . . .	788
9.3.2.1	Simplified SDL syntax . . . . .	788
9.3.2.2	Query performance analysis . . . . .	789
9.3.2.3	UI improvements . . . . .	790
9.3.2.3.1	New UI for setting globals and configuration . . . . .	790
9.3.2.3.2	New UI REPL . . . . .	790
9.3.2.3.3	Query editor and visual builder . . . . .	791
9.3.2.4	<code>edgedb watch</code> and a new development workflow . . . . .	792
9.3.2.4.1	1. Start the <code>watch</code> command . . . . .	792
9.3.2.4.2	2. Write an initial schema . . . . .	792
9.3.2.4.3	3. Edit your schema files . . . . .	792
9.3.2.4.4	4. Generate a migration . . . . .	792
9.3.2.5	Triggers . . . . .	793
9.3.2.6	Mutation rewrites . . . . .	793
9.3.2.7	Splats . . . . .	793
9.3.2.8	SQL support . . . . .	794
9.3.2.9	Nested modules . . . . .	795
9.3.2.10	<code>intersect</code> and <code>except</code> operators . . . . .	795
9.3.2.11	<code>assert</code> function . . . . .	796
9.3.3	Additional changes . . . . .	796
9.3.3.1	EdgeQL . . . . .	796
9.3.3.2	CLI . . . . .	798
9.3.3.3	Bug fixes . . . . .	798
9.3.3.4	Deprecations . . . . .	799
9.3.4	New release schedule . . . . .	799
9.3.5	3.0 RC 1 . . . . .	799
9.3.5.1	Changes to new 3.0 features . . . . .	799
9.3.5.2	Other changes and fixes . . . . .	800
9.4	Deprecation Policy . . . . .	800

Welcome to the EdgeDB 0.5.0 documentation.



---

CHAPTER  
ONE

---

## GET STARTED

### 1.1 Quickstart

Welcome to EdgeDB!

This quickstart will walk you through the entire process of creating a simple EdgeDB-powered application: installation, defining your schema, adding some data, and writing your first query. Let's jump in!

#### 1.1.1 1. Installation

First let's install the EdgeDB CLI. Open a terminal and run the appropriate command below.

##### macOS/Linux

```
$ curl https://sh.edgedb.com --proto '=https' -sSf1 | sh
```

##### Windows (Powershell)

---

**Note:** EdgeDB on Windows requires WSL 2 because the EdgeDB server runs on Linux.

---

```
PS> iwr https://ps1.edgedb.com -useb | iex
```

This command downloads and executes a bash script that installs the edgedb CLI on your machine. You may be asked for your password. Once the installation completes, you may need to **restart your terminal** before you can use the edgedb command.

---

**Note:** Check out our additional installation methods [for various Linux distros](#), via [Homebrew on macOS](#), and [for the Windows Command Prompt](#).

---

Now let's set up your EdgeDB project.

### 1.1.2 2. Initialize a project

In a terminal, create a new directory and cd into it.

```
$ mkdir quickstart  
$ cd quickstart
```

Then initialize your EdgeDB project:

```
$ edgedb project init
```

This starts an interactive tool that walks you through the process of setting up your first EdgeDB instance. You should see something like this:

```
$ edgedb project init  
No `edgedb.toml` found in `/path/to/quickstart` or above  
Do you want to initialize a new project? [Y/n]  
> Y  
Specify the name of EdgeDB instance to use with this project [quickstart]:  
> quickstart  
Checking EdgeDB versions...  
Specify the version of EdgeDB to use with this project [default: 2.x]:  
> 2.x
```

Project directory	~/path/to/quickstart
Project config	~/path/to/quickstart/edgedb.toml
Schema dir (empty)	~/path/to/quickstart/dbschema
Installation method	portable package
Version	2.x+c21decd
Instance name	quickstart

```
Downloading package...  
00:00:01 [=====] 32.98MiB/32.98MiB 32.89MiB/s | ETA: 0s  
Successfully installed 2.x+c21decd  
Initializing EdgeDB instance...  
Applying migrations...  
Everything is up to date. Revision initial  
Project initialized.  
To connect to quickstart, run `edgedb`
```

This did a couple things.

1. First, it scaffolded your project by creating an `edgedb.toml` config file and a schema file `dbschema/default.esdl`. In the next section, you'll define a schema in `default.esdl`.
2. Second, it spun up an EdgeDB instance called `quickstart` and “linked” it to the current directory. As long as you're inside the project directory, all CLI commands will be executed against this instance. For more details on how EdgeDB projects work, check out the [Managing instances](#) guide.

---

**Note:** Quick note! You can have several **instances** of EdgeDB running on your computer simultaneously. Each instance contains several **databases**. Each database may contain several **modules** (though commonly your schema will be entirely defined inside the `default` module).

---

Let's connect to our new instance! Run `edgedb` in your terminal to open an interactive REPL to your instance. You're now connected to a live EdgeDB instance running on your computer! Try executing a simple query:

```
db> select 1 + 1;
{2}
```

Run `\q` to exit the REPL. More interesting queries are coming soon, promise! But first we need to set up a schema.

### 1.1.3 3. Set up your schema

Open the `quickstart` directory in your IDE or editor of choice. You should see the following file structure.

```
/path/to/quickstart
└── edgedb.toml
└── dbschema
    ├── default.esdl
    └── migrations
```

EdgeDB schemas are defined with a dedicated schema description language called (predictably) EdgeDB SDL (or just **SDL** for short). It's an elegant, declarative way to define your data model.

SDL lives inside `.esdl` files. Commonly, your entire schema will be declared in a file called `default.esdl` but you can split your schema across several `.esdl` files if you prefer.

---

**Note:** Syntax-highlighter packages/extensions for `.esdl` files are available for [Visual Studio Code](#), [Sublime Text](#), [Atom](#), and [Vim](#).

---

Let's build a simple movie database. We'll need to define two **object types** (equivalent to a *table* in SQL): `Movie` and `Person`. Open `dbschema/default.esdl` in your editor of choice and paste the following:

```
module default {
    type Person {
        required property name -> str;
    }

    type Movie {
        property title -> str;
        multi link actors -> Person;
    }
};
```

```
module default {
    type Person {
        required name: str;
    }

    type Movie {
        title: str;
        multi actors: Person;
    }
};
```

A few things to note here.

- Our types don't contain an `id` property; EdgeDB automatically creates this property and assigns a unique UUID to every object inserted into the database.

- The `Movie` type includes a `link` named `actors`. In EdgeDB, links are used to represent relationships between object types. They eliminate the need for foreign keys; later, you'll see just how easy it is to write "deep" queries without `JOINS`.
- The object types are inside a `module` called `default`. You can split up your schema into logical subunits called `modules`, though it's common to define the entire schema in a single module called `default`.

Now we're ready to run a migration to apply this schema to the database.

### 1.1.4 4. Run a migration

Generate a migration file with `edgedb migration create`. This command gathers up our `*.esdl` files and sends them to the database. The *database itself* parses these files, compares them against its current schema, and generates a migration plan! Then the database sends this plan back to the CLI, which creates a migration file.

```
$ edgedb migration create
Created ./dbschema/migrations/00001.edgeql (id: <hash>)
```

---

**Note:** If you're interested, open this migration file to see what's inside! It's a simple EdgeQL script consisting of `DDL` commands like `create type`, `alter type`, and `create property`.

---

The migration file has been *created* but we haven't *applied* it against the database. Let's do that.

```
$ edgedb migrate
Applied m1k54jubcs62wlzfebn3pxwwngajv1bf6c6qfs1suagky1g2fzv2lq (00001.edgeql)
```

Looking good! Let's make sure that worked by running `edgedb list types` on the command line. This will print a table containing all currently-defined object types.

```
$ edgedb list types
  _____
  | Name      | Extending |
  | default::Movie | std::BaseObject, std::Object |
  | default::Person | std::BaseObject, std::Object |
```

Before we proceed, let's try making a small change to our schema: making the `title` property of `Movie` required. First, update the schema file:

```
type Movie {
-   property title -> str;
+   required property title -> str;
    multi link actors -> Person;
}
```

```
type Movie {
-   title: str;
+   required title: str;
    multi actors: Person;
}
```

Then create another migration. Because this isn't the initial migration, we see something a little different than before.

```
$ edgedb migration create
did you make property 'title' of object type 'default::Movie'
required? [y,n,l,c,b,s,q,?]
>
```

As before, EdgeDB parses the schema files and compared them against its current internal schema. It correctly detects the change we made, and prompts us to confirm it. This interactive process lets you sanity check every change and provide guidance when a migration is ambiguous (e.g. when a property is renamed).

Enter `y` to confirm the change.

```
$ edgedb migration create
did you make property 'title' of object type 'default::Movie'
required? [y,n,l,c,b,s,q,?]
> y
Please specify an expression to populate existing objects in
order to make property 'title' of object type 'default::Movie' required:
fill_expr>
```

Hm, now we're seeing another prompt. Because `title` is changing from *optional* to *required*, EdgeDB is asking us what to do for all the `Movie` objects that don't currently have a value for `title` defined. We'll just specify a placeholder value: "Untitled".

```
fill_expr> "Untitled"
Created dbschema/migrations/00002.edgeql (id: <hash>)
```

If we look at the generated migration file, we see it contains the following lines:

```
ALTER TYPE default::Movie {
    ALTER PROPERTY title {
        SET REQUIRED USING ("Untitled");
    };
};
```

Let's wrap up by applying the new migration.

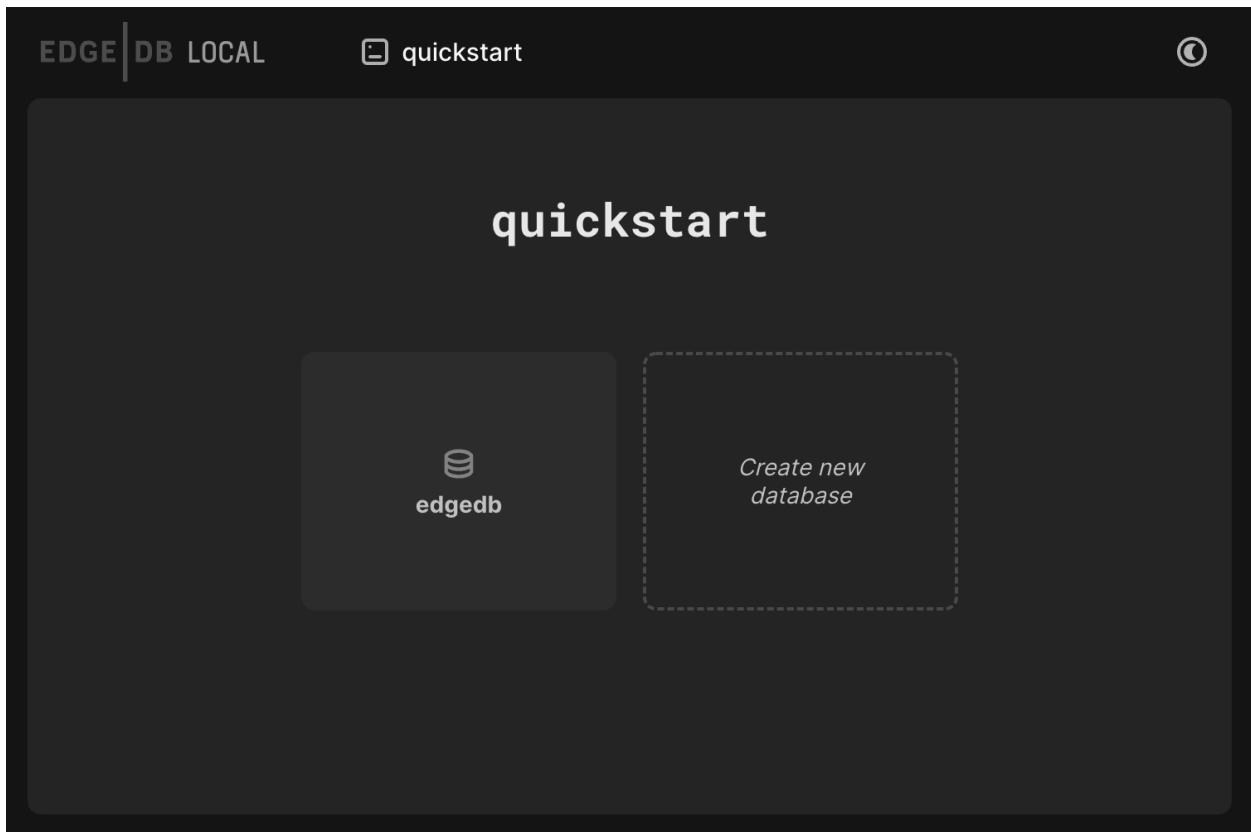
```
$ edgedb migrate
Applied m1rd2ikgwdtlj5ws71l6rwzvyiui2xbrkzig4adsvwy2sje7kxeh3a (00002.edgeql)
```

### 1.1.5 5. Write some queries

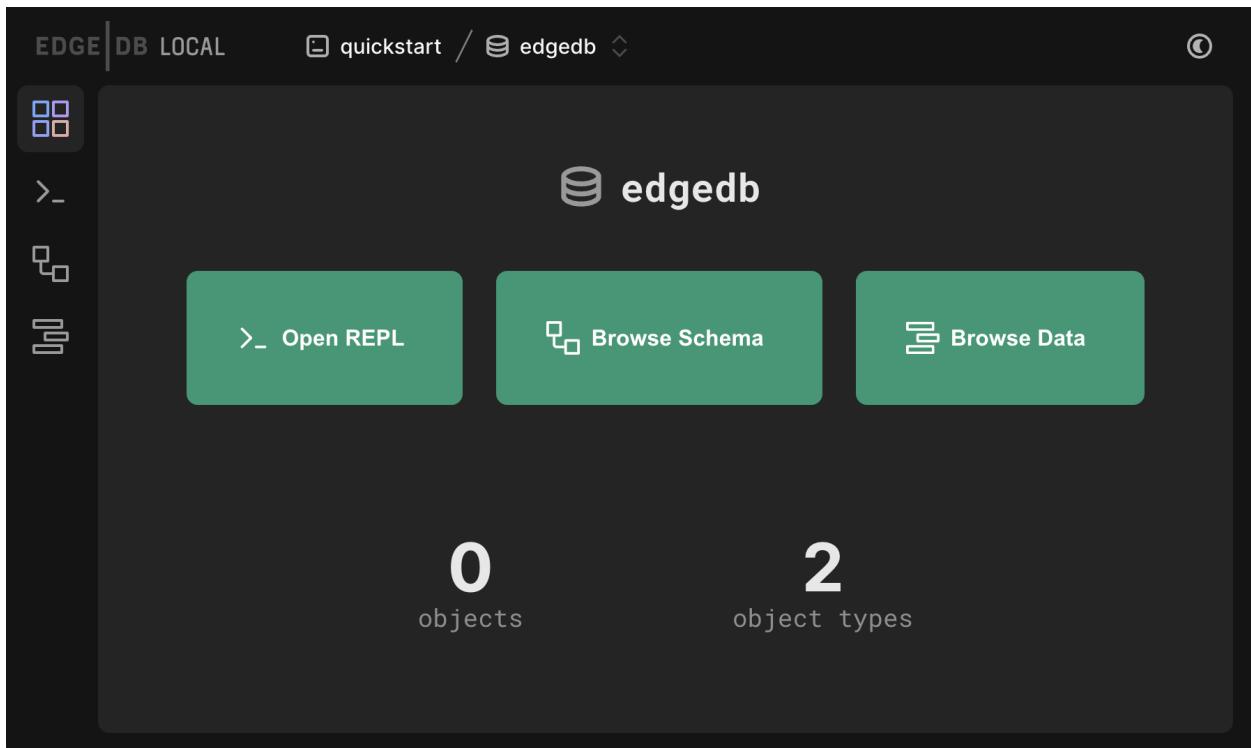
Let's write some simple queries via *EdgeDB UI*, the admin dashboard baked into every EdgeDB instance (v2.0+ only). To open the dashboard:

```
$ edgedb ui
Opening URL in browser:
http://localhost:107xx/ui?authToken=<jwt token>
```

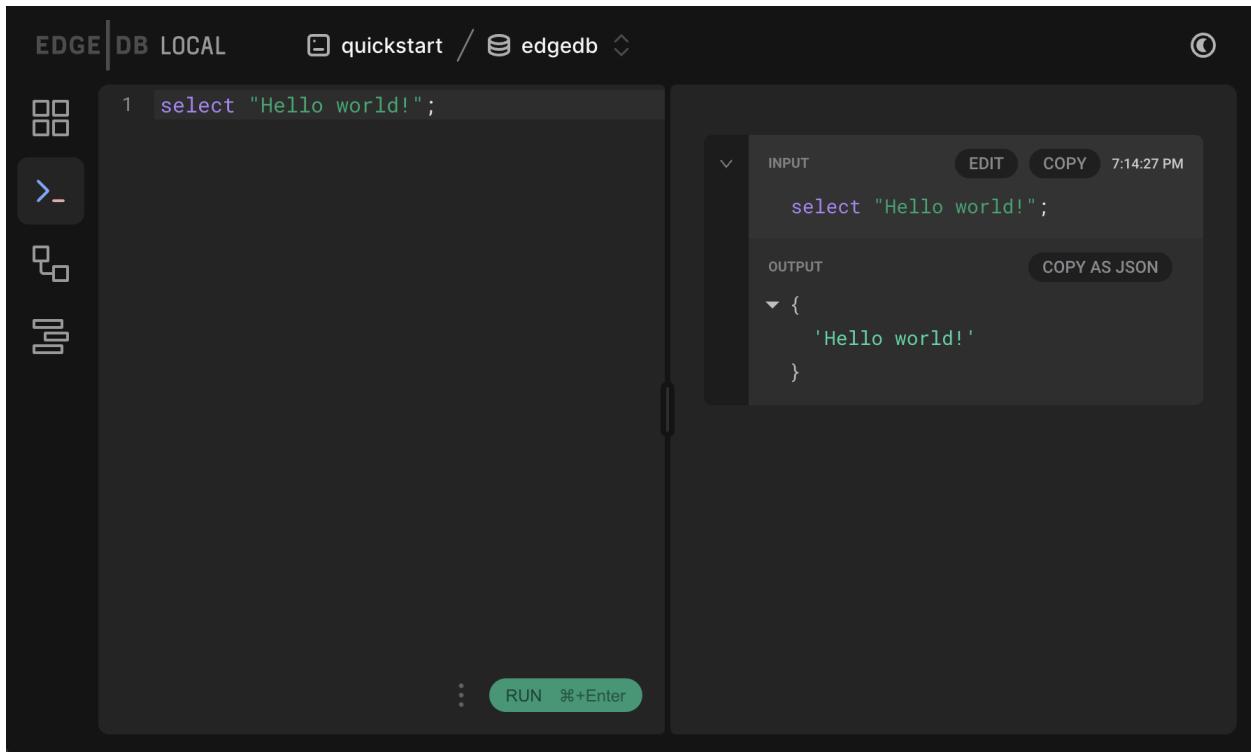
You should see a simple landing page, as below. You'll see a card for each database running on your instance—remember: each instance can contain multiple databases!



Currently, there's only one database, which is simply called `edgedb` by default. Click the `edgedb` card.



Then click Open REPL so we can start writing some queries. We'll start simple: `select "Hello world!"`. Click RUN to execute the query.



The query should appear in the “query notebook” on the right, along with the result of the query.

Now let’s actually `insert` an object into our database. Copy the following query into the query textarea and hit Run.

```
insert Movie {
    title := "Dune"
};
```

Nice! You’ve officially inserted the first object into your database! Let’s add a couple cast members with an `update` query.

```
update Movie
filter .title = "Dune"
set {
    actors := {
        (insert Person { name := "Timothée Chalamet" }),
        (insert Person { name := "Zendaya" })
    }
};
```

Finally, we can run a `select` query to fetch all the data we just inserted.

```
select Movie {
    title,
    actors: {
        name
    }
};
```

Click `COPY AS JSON` to copy the result of this query to your clipboard. It will look something like this:

```
[  
  {  
    "title": "Dune",  
    "actors": [  
      { "name": "Timothée Chalamet" },  
      { "name": "Zendaya" }  
    ]  
  }  
]
```

EdgeDB UI is a useful development tool, but in practice your application will likely be using one of EdgeDB's *client libraries* to execute queries. EdgeDB provides official libraries for [JavaScript/TypeScript](#), [Go](#), [Python](#), [Rust](#), and [C#](#) and [F#](#). Check out the [Clients](#) guide to get started with the language of your choice.

### 1.1.6 Onwards and upwards

You now know the basics of EdgeDB! You've installed the CLI and database, set up a local project, run a couple migrations, inserted and queried some data, and used a client library.

- For a more in-depth exploration of each topic covered here, continue reading the other pages in the Getting Started section, which will cover important topics like migrations, the schema language, and EdgeQL in greater detail.
- For guided tours of major concepts, check out the showcase pages for [Data Modeling](#), [EdgeQL](#), and [Migrations](#).
- For a deep dive into the EdgeQL query language, check out the [Interactive Tutorial](#).
- For an immersive, comprehensive walkthrough of EdgeDB concepts, check out our illustrated e-book [Easy EdgeDB](#); it's designed to walk a total beginner through EdgeDB, from the basics all the way through advanced concepts.
- To start building an application using the language of your choice, check out our client libraries for [JavaScript/TypeScript](#), [Python](#), and [Go](#).
- Or just jump into the *docs*!

## 1.2 The CLI

The `edgedb` command line tool is an integral part of the developer workflow of building with EdgeDB. Below are instructions for installing it.

### 1.2.1 Installation

To get started with EdgeDB, the first step is install the `edgedb` CLI.

**Linux or macOS**

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.edgedb.com | sh
```

**Windows Powershell**

---

**Note:** EdgeDB on Windows requires WSL 2 because the EdgeDB server runs on Linux.

---

```
PS> iwr https://ps1.edgedb.com -useb | iex
```

Follow the prompts on screen to complete the installation. The script will download the `edgedb` command built for your OS and add a path to it to your shell environment. Then test the installation:

```
$ edgedb --version
EdgeDB CLI 2.x+abcdefg
```

---

**Note:** If you encounter a `command not found` error, you may need to open a fresh shell window.

---



---

**Note:** To install the CLI with a package manager, refer to the “Additional methods” section of the [Install](#) page for instructions.

---

### 1.2.2 See help commands

The entire CLI is self-documenting. Once it’s installed, run `edgedb --help` to see a breakdown of all the commands and options.

```
$ edgedb --help
EdgeDB CLI
Use the edgedb command-line tool to spin up local instances, manage EdgeDB projects, create and apply migrations, and more.

Running edgedb without a subcommand opens an interactive shell.

USAGE:
  edgedb [OPTIONS] [SUBCOMMAND]

OPTIONS:
  <list of options>

CONNECTION OPTIONS (edgedb --help-connect to see the full list):
  <list of connection options>

SUBCOMMANDS:
  <list of all major commands>
```

The majority of CLI commands perform some action against a *particular* EdgeDB instance. As such, there are a standard set of flags that are used to specify *which instance* should be the target of the command, plus additional information like TLS certificates. The following command documents these flags.

```
$ edgedb --help-connect
-I, --instance <instance>
  Local instance name created with edgedb instance create to connect to
  (overrides host and port)
--dsn <dsn>
  DSN for EdgeDB to connect to (overrides all other options except
  password)
--credentials-file <credentials_file>
```

(continues on next page)

(continued from previous page)

```
Path to JSON file to read credentials from
-H, --host <host>
    Host of the EdgeDB instance
-P, --port <port>
    Port to connect to EdgeDB
--unix-path <unix_path>
    Unix socket dir for the
-u, --user <user>
    User name of the EdgeDB user
-d, --database <database>
    Database name to connect to
--password
    Ask for password on the terminal (TTY)
--no-password
    Don't ask for password
```

If you ever want to see documentation for a particular command (`edgedb migration create`) or group of commands (`edgedb instance`), just append the `--help` flag.

```
$ edgedb instance --help
Manage local EdgeDB instances

USAGE:
    edgedb instance <SUBCOMMAND>

OPTIONS:
    -h, --help      Print help information

SUBCOMMANDS:
    create          Initialize a new EdgeDB instance
    credentials     Echo credentials to connect to the instance
    destroy         Destroy an instance and remove the data
    link            Link a remote instance
    list            Show all instances
    ...
```

### 1.2.3 Upgrade the CLI

To upgrade to the latest version:

```
$ edgedb cli upgrade
```

## 1.3 Instances

Let's get to the good stuff. You can spin up an EdgeDB instance with a single command.

```
$ edgedb instance create my_instance
```

This creates a new instance named `my_instance` that runs the latest stable version of EdgeDB. (EdgeDB itself will be automatically installed if it isn't already.) Alternatively you can specify a specific version with `--version`.

```
$ edgedb instance create my_instance --version 2.1
$ edgedb instance create my_instance --version nightly
```

We can execute a query against our new instance with `edgedb query`. Specify which instance to connect to by passing an instance name into the `-I` flag.

```
$ edgedb query "select 3.14" -I my_instance
3.14
```

### 1.3.1 Creating databases

A single EdgeDB *instance* can contain multiple *databases*. Upon creation, an instance contains a single database called `edgedb`. All queries and CLI commands are executed against this database unless otherwise specified.

To create a new database:

```
$ edgedb database create newdb -I my_instance
```

We can now execute queries against this new database by specifying it with the `--database/-d` flag.

```
$ edgedb query "select 3.14" -I my_instance -d newdb
3.14
```

### 1.3.2 Managing instances

Instances can be stopped, started, restarted, and destroyed.

```
$ edgedb instance stop -I my_instance
$ edgedb instance start -I my_instance
$ edgedb instance restart -I my_instance
$ edgedb instance destroy -I my_instance
```

### 1.3.3 Listing instances

To list all instances on your machine:

```
$ edgedb instance list
```

Kind	Name	Port	Version	Status	
local	my_instance	10700	2.x+8421216	active	

(continues on next page)

(continued from previous page)

local	my_instance_2	10701	2.x+8421216	active	
local	my_instance_3	10702	2.x+8421216	active	

### 1.3.4 Further reference

For complete documentation on managing instances with the CLI (upgrading, viewing logs, etc.), refer to the [edgedb instance](#) reference or view the help text in your shell:

```
$ edgedb instance --help
```

## 1.4 Projects

It can be inconvenient to pass the `-I` flag every time you wish to run a CLI command.

```
$ edgedb migration create -I my_instance
```

That's one of the reasons we introduced the concept of an *EdgeDB project*. A project is a directory on your file system that is associated ("linked") with an EdgeDB instance.

---

**Note:** Projects are intended to make *local development* easier! They only exist on your local machine and are managed with the CLI. When deploying EdgeDB for production, you will typically pass connection information to the client library using environment variables.

---

When you're inside a project, all CLI commands will be applied against the *linked instance* by default (no CLI flags required).

```
$ edgedb migration create
```

The same is true for all EdgeDB client libraries (discussed in more depth in the [Clients](#) section). If the following file lives inside an EdgeDB project directory, `createClient` will discover the project and connect to its linked instance with no additional configuration.

```
// clientTest.js
import {createClient} from 'edgedb';

const client = createClient();
await client.query("select 5");
```

### 1.4.1 Initializing

To initialize a project, create a new directory and run `edgedb project init` inside it. You'll see something like this:

```
$ edgedb project init
No `edgedb.toml` found in this repo or above.
Do you want to initialize a new project? [Y/n]
> Y
Specify the version of EdgeDB to use with this project [2.x]:
> # (left blank for default)
Specify the name of EdgeDB instance to use with this project:
> my_instance
Initializing EdgeDB instance...
Bootstrap complete. Server is up and running now.
Project initialized.
```

This command does a couple important things.

1. It spins up a new EdgeDB instance called `my_instance`.
2. If no `edgedb.toml` file exists, it will create one. This is a configuration file that indicates that a given directory is an EdgeDB project. Currently it only supports a single setting: `server-version`.

<b>[edgedb]</b>
server-version = "2.1"

3. If no `dbschema` directory exists, it will be created, along with an empty `default.esdl` file which will contain your schema. If a `dbschema` directory exists and contains a subdirectory called `migrations`, those migrations will be applied against the new instance.

Every project maps one-to-one to a particular EdgeDB instance. From inside a project directory, you can run `edgedb project info` to see information about the current project.

\$ edgedb project info	Instance name   my_instance	Project root   /path/to/project
------------------------	-----------------------------	---------------------------------

### 1.4.2 Connection

As long as you are inside the project directory, all CLI commands will be executed against the project-linked instance. For instance, you can simply run `edgedb` to open a REPL.

```
$ edgedb
EdgeDB 2.x+88c1706 (repl 2.x+a7fc49b)
Type \help for help, \quit to quit.
edgedb> select "Hello world!";
```

By contrast, if you leave the project directory, the CLI will no longer know which instance to connect to. You can solve this by specifying an instance name with the `-I` flag.

```
$ cd ~
$ edgedb
```

(continues on next page)

(continued from previous page)

```
ClientNoCredentialsError: no `edgedb.toml` found and no
connection options are specified
Hint: Run `edgedb project init` or use any of `-H`, `-P`, `-I` arguments
to specify connection parameters. See `--help` for details
$ edgedb -I my_instance
EdgeDB 2.x+88c1706 (repl 2.x+a7fc49b)
Type \help for help, \quit to quit.
edgedb>
```

Similarly, client libraries will auto-connect to the project's linked instance without additional configuration.

### 1.4.3 Using remote instances

You may want to initialize a project that points to a remote EdgeDB instance. This is totally a valid case and EdgeDB fully supports it! Before running `edgedb project init`, you just need to create an alias for the remote instance using `edgedb instance link`, like so:

```
$ edgedb instance link
Specify the host of the server [default: localhost]:
> 192.168.4.2
Specify the port of the server [default: 5656]:
> 10818
Specify the database user [default: edgedb]:
> edgedb
Specify the database name [default: edgedb]:
> edgedb
Unknown server certificate: SHA1:c38a7a90429b033dfaf7a81e08112a9d58d97286.
Trust? [y/N]
> y
Password for 'edgedb':
Specify a new instance name for the remote server [default: abcd]:
> staging_db
Successfully linked to remote instance. To connect run:
  edgedb -I staging_db
```

After receiving the necessary connection information, this command links the remote instance to a local alias "staging\_db". You can use this as instance name in CLI commands.

```
$ edgedb -I staging_db
edgedb>
```

To initialize a project that uses the remote instance, provide this alias when prompted for an instance name during the `edgedb project init` workflow.

## 1.4.4 Unlinking

An instance can be unlinked from a project. This leaves the instance running but effectively “uninitializes” the project. The `edgedb.toml` and `dbschema` are left untouched.

```
$ edgedb project unlink
```

If you wish to delete the instance as well, use the `-D` flag.

```
$ edgedb project unlink -D
```

## 1.4.5 Upgrading

A standalone instance (not linked to a project) can be upgraded with the `edgedb instance upgrade` command.

```
$ edgedb project upgrade --to-latest
$ edgedb project upgrade --to-nightly
$ edgedb project upgrade --to-version 2.x
```

## 1.4.6 See info

You can see the location of a project and the name of its linked instance.

```
$ edgedb project info
```

Instance name	<code>my_app</code>
Project root	<code>/path/to/my_app</code>

## 1.5 Schema

This page is intended as a rapid-fire overview of EdgeDB’s schema definition language (SDL) so you can hit the ground running with EdgeDB. Refer to the linked pages for more in-depth documentation!

### 1.5.1 Scalar types

EdgeDB implements a rigorous type system containing the following primitive types.

Strings	<code>str</code>
Booleans	<code>bool</code>
Numbers	<code>int32 int64 float32 float64 bigint decimal</code>
UUID	<code>uuid</code>
JSON	<code>json</code>
Dates and times	<code>datetime cal::local_datetime cal::local_date cal::local_time</code>
Durations	<code>duration cal::relative_duration cal::date_duration</code>
Binary data	<code>bytes</code>
Auto-incrementing counters	<code>sequence</code>
Enums	<code>enum&lt;x, y, z&gt;</code>

These primitives can be combined into arrays, tuples, and ranges.

Arrays	<code>array&lt;str&gt;</code>
Tuples (unnamed)	<code>tuple&lt;str, int64, bool&gt;</code>
Tuples (named)	<code>tuple&lt;name: str, age: int64, is_awesome: bool&gt;</code>
Ranges	<code>range&lt;float64&gt;</code>

Collectively, *primitive* and *collection* types comprise EdgeDB's *scalar type system*.

## 1.5.2 Object types

Object types are analogous to tables in SQL. They can contain **properties**—which can correspond to any scalar type—and **links**—which correspond to other object types.

## 1.5.3 Properties

The `property` keyword is used to declare a property.

```
type Movie {  
    property title -> str;  
}
```

```
type Movie {  
    title: str;  
}
```

See *Schema > Object types*.

### 1.5.3.1 Required vs optional

Properties are optional by default. Use the `required` keyword to make them required.

```
type Movie {  
    required property title -> str;      # required  
    property release_year -> int64;       # optional  
}
```

```
type Movie {  
    required title: str;      # required  
    release_year: int64;       # optional  
}
```

See *Schema > Properties*.

### 1.5.3.2 Constraints

Add a pair of curly braces after the property to define additional information, including constraints.

```
type Movie {
    required property title -> str {
        constraint exclusive;
        constraint min_len_value(8);
        constraint regexp(r'^[A-Za-z0-9 ]+$');
    }
}
```

```
type Movie {
    required title: str {
        constraint exclusive;
        constraint min_len_value(8);
        constraint regexp(r'^[A-Za-z0-9 ]+$');
    }
}
```

See [Schema > Constraints](#).

### 1.5.3.3 Computed properties

Object types can contain *computed properties* that correspond to EdgeQL expressions. This expression is dynamically computed whenever the property is queried.

```
type Movie {
    required property title -> str;
    property uppercase_title := str_upper(.title);
}
```

```
type Movie {
    required title: str;
    property uppercase_title := str_upper(.title);
}
```

See [Schema > Computeds](#).

## 1.5.4 Links

Object types can have links to other object types.

```
type Movie {
    required property title -> str;
    link director -> Person;
}

type Person {
    required property name -> str;
}
```

```
type Movie {
    required title: str;
    director: Person;
}

type Person {
    required name: str;
}
```

Use the `required` and `multi` keywords to specify the cardinality of the relation.

```
type Movie {
    required property title -> str;

    link cinematographer -> Person;                      # zero or one
    required link director -> Person;                     # exactly one
    multi link writers -> Person;                        # zero or more
    required multi link actors -> Person;                 # one or more
}

type Person {
    required property name -> str;
}
```

```
type Movie {
    required title: str;

    cinematographer: Person;                            # zero or one
    required director: Person;                         # exactly one
    multi writers: Person;                           # zero or more
    required multi actors: Person;                    # one or more
}

type Person {
    required name: str;
}
```

To define a one-to-one relation, use an `exclusive` constraint.

```
type Movie {
    required property title -> str;
    required link stats -> MovieStats {
        constraint exclusive;
    };
}

type MovieStats {
    required property budget -> int64;
    required property box_office -> int64;
}
```

```
type Movie {
    required title: str;
```

(continues on next page)

(continued from previous page)

```

required stats: MovieStats {
    constraint exclusive;
};

type MovieStats {
    required budget: int64;
    required box_office: int64;
}

```

See [Schema > Links](#).

#### 1.5.4.1 Computed links

Objects can contain “computed links”: stored expressions that return a set of objects. Computed links are dynamically computed when they are referenced in queries. The example below defines a backlink.

```

type Movie {
    required property title -> str;
    multi link actors -> Person;

    # returns all movies with same title
    multi link same_title := (
        with t := .title
        select detached Movie filter .title = t
    )
}

```

```

type Movie {
    required title: str;
    multi actors: Person;

    # returns all movies with same title
    multi link same_title := (
        with t := .title
        select detached Movie filter .title = t
    )
}

```

#### 1.5.4.2 Backlinks

A common use case for computed links is *backlinks*.

```

type Movie {
    required property title -> str;
    multi link actors -> Person;
}

type Person {
    required property name -> str;
}

```

(continues on next page)

(continued from previous page)

```
multi link acted_in := .<actors[is Movie];
}
```

```
type Movie {
    required title: str;
    multi actors: Person;
}

type Person {
    required name: str;
    multi link acted_in := .<actors[is Movie];
}
```

The computed link `acted_in` returns all `Movie` objects with a link called `actors` that points to the current `Person`. The easiest way to understand backlink syntax is to split it into two parts:

**.<actors** This uses a special syntax `.<` to return all objects in the database with a link called `actors` that points to the current object. This set could conceivably contain other objects besides `Movie`; for instance, we could define a `TVShow` type that also included link `actors -> Person`.

**[is Movie]** This is a *type filter* that filters out all objects that aren't `Movie` objects. A backlink still works without this filter, but could contain any other number of objects besides ``Movie`` objects.

See [Schema > Computeds > Backlinks](#).

## 1.5.5 Constraints

Constraints can also be defined at the *object level*.

```
type BlogPost {
    property title -> str;
    link author -> User;

    constraint exclusive on ((.title, .author));
}
```

```
type BlogPost {
    title: str;
    author: User;

    constraint exclusive on ((.title, .author));
}
```

Constraints can contain exceptions; these are called *partial constraints*.

```
type BlogPost {
    property title -> str;
    property published -> bool;

    constraint exclusive on (.title) except (not .published);
}
```

```
type BlogPost {
    title: str;
    published: bool;

    constraint exclusive on (.title) except (not .published);
}
```

## 1.5.6 Indexes

Use `index on` to define indexes on an object type.

```
type Movie {
    required property title -> str;
    required property release_year -> int64;

    index on (.title);                      # simple index
    index on ((.title, .release_year));       # composite index
    index on (str_trim(str_lower(.title)));   # computed index
}
```

```
type Movie {
    required title: str;
    required release_year: int64;

    index on (.title);                      # simple index
    index on ((.title, .release_year));       # composite index
    index on (str_trim(str_lower(.title)));   # computed index
}
```

The `id` property, all links, and all properties with `exclusive` constraints are automatically indexed.

See [Schema > Indexes](#).

## 1.5.7 Schema mixins

Object types can be declared as `abstract`. Non-abstract types can `extend` abstract types.

```
abstract type Content {
    required property title -> str;
}

type Movie extending Content {
    required property release_year -> int64;
}

type TVShow extending Content {
    required property num_seasons -> int64;
}
```

```
abstract type Content {
    required title: str;
```

(continues on next page)

(continued from previous page)

```

}

type Movie extending Content {
    required release_year: int64;
}

type TVShow extending Content {
    required num_seasons: int64;
}

```

Multiple inheritance is supported.

```

abstract type HasTitle {
    required property title -> str;
}

abstract type HasReleaseYear {
    required property release_year -> int64;
}

type Movie extending HasTitle, HasReleaseYear {
    link sequel_to -> Movie;
}

```

```

abstract type HasTitle {
    required title: str;
}

abstract type HasReleaseYear {
    required release_year: int64;
}

type Movie extending HasTitle, HasReleaseYear {
    sequel_to: Movie;
}

```

See [Schema > Object types > Inheritance](#).

## 1.5.8 Polymorphism

Links can correspond to abstract types. These are known as *polymorphic links*.

```

abstract type Content {
    required property title -> str;
}

type Movie extending Content {
    required property release_year -> int64;
}

type TVShow extending Content {

```

(continues on next page)

(continued from previous page)

```
required property num_seasons -> int64;
}

type Franchise {
  required property name -> str;
  multi link entries -> Content;
}
```

```
abstract type Content {
  required title: str;
}

type Movie extending Content {
  required release_year: int64;
}

type TVShow extending Content {
  required num_seasons: int64;
}

type Franchise {
  required name: str;
  multi entries: Content;
}
```

See [Schema > Links > Polymorphism](#) and [EdgeQL > Select > Polymorphic queries](#).

## 1.6 Migrations

**index** migrations fill\_expr cast\_expr

EdgeDB's baked-in migration system lets you painlessly evolve your schema throughout the development process. If you want to work along with this guide, start a new project with `edgedb project init`. This will create a new instance and create some empty schema files to get you started.

The recommended workflow has been improved in version 3.0, so if you're running our beta, try the new workflow instead.

### 1.6.1 1. Write an initial schema

By convention, your EdgeDB schema is defined inside one or more `.esdl` files that live in a directory called `dbschema` in the root directory of your codebase.

```
.
└── dbschema
    └── default.esdl          # schema file (written by you)
        └── migrations         # migration files (typically generated by CLI)
            └── 00001.edgeql
            └── ...
└── edgedb.toml
```

The schema itself is written using EdgeDB's schema definition language.

```
type User {
    required property name -> str;
}

type Post {
    required property title -> str;
    required link author -> User;
}
```

```
type User {
    required name: str;
}

type Post {
    required title: str;
    required author: User;
}
```

It's common to keep your entire schema in a single file, typically called `default.esdl`. However it's also possible to split it across a number of `.esdl` files.

To spin up a new instance and populate it with an initial schema, execute the following commands in a fresh directory.

```
$ edgedb project init
Do you want to initialize a new project? [Y/n]
> Y
<additional prompts>
$ edgedb migration create
Created dbschema/migrations/00001.edgeql (id: <hash>)
$ edgedb migrate
Applied dbschema/migrations/00001.edgeql (id: <hash>)
```

## 1.6.2 2. Edit your schema files

As your application evolves, directly edit your schema files to reflect your desired data model.

```
type User {
    required property name -> str;
}

type BlogPost {
    property title -> str;
    required link author -> User;
}

+ type Comment {
+     required property content -> str;
+ }
```

```

type User {
    required name: str;
}

type BlogPost {
    title: str;
    required author: User;
}

+ type Comment {
+   required content: str;
+ }

```

### 1.6.3 3. Generate a migration

To generate a migration that reflects these changes, run `edgedb migration create`.

```
$ edgedb migration create
```

The CLI reads your schema file and sends it to the active EdgeDB instance. The instance compares the file's contents to its current schema state and determines a migration plan. **The migration plan is generated by the database itself.**

This plan is then presented to you interactively; each detected schema change will be individually presented to you for approval. For each prompt, you have a variety of commands at your disposal. Type `y` to approve, `n` to reject, `q` to cancel the migration, or `?` for a breakdown of some more advanced options.

```

$ edgedb migration create
Did you create object type 'default::Comment'? [y,n,l,c,b,s,q,?]
> y
Created dbschema/migrations/00002.edgeql (id: <hash>)

```

### 1.6.4 4. Apply the migration

We've generated a migration file, but we haven't yet applied it against our database! The following command will apply all unapplied migration files:

```
$ edgedb migrate
Applied m1virjowa... (00002.edgeql)
```

That's it! You've created and applied your first EdgeDB migration. Your instance is now using the latest schema.

### 1.6.5 Data migrations

Depending on how the schema was changed, you may be prompted to provide an EdgeQL expression to map the contents of your database to the new schema. To see this happen, let's make the `title` property required.

```

type User {
    required property name -> str;
}

```

(continues on next page)

(continued from previous page)

```
type BlogPost {
-   property title -> str;
+   required property title -> str;
    required link author -> User;
}
```

```
type User {
    required name: str;
}

type BlogPost {
-   title: str;
+   required title: str;
    required author: User;
}
```

Then we'll create another migration.

```
$ edgedb migration create
Did you make property 'title' of object type
'default::BlogPost' required? [y,n,l,c,b,s,q,?]
> y
Please specify an expression to populate existing objects in order to make
property 'title' of object type 'default::Post' required:
fill_expr>
```

Because `title` is currently optional, the database may contain blog posts without a `title` property. The expression you provide will be used to *assign a title* to any post that doesn't have one. We'll just provide a simple default title: '`Untitled`'.

```
fill_expr> 'Untitled'
Created dbschema/migrations/00002.edgeql, id:
m1yt3gbstvyfzy2rhqt53351d6br2amw7ywqu2bvjiqsacbcdxzyya
```

Nice! It accepted our answer and created a new migration file `00002.edgeql`. Let's see what the newly created `00002.edgeql` file contains.

```
CREATE MIGRATION m1yt3gbstvyfzy2rhqt53351d6br2amw7ywqu2bvjiqsacbcdxzyya
  ONTO m1cvx47vntfoy24evwrdli7o5unaxc2c5t3i2rfspd2qosi6d6iahq
{
  ALTER TYPE default::Post {
    ALTER PROPERTY title {
      SET REQUIRED USING ('Untitled');
    };
  };
};
```

We have a `CREATE MIGRATION` block containing an `ALTER TYPE` statement to make `Post.title` required. We can see that our fill expression ('`Untitled`') is included directly in the migration file.

Note that we could have provided an *arbitrary EdgeQL expression!* The following EdgeQL features are often useful:

<code>assert_exists</code>	This is an “escape hatch” function that tells EdgeDB to assume the input has <i>at least</i> one element.  <code>fill_expr&gt; assert_exists(.title)</code> If you provide a <code>fill_expr</code> like the one above, you must separately ensure that all movies have a title before executing the migration; otherwise it will fail.
<code>assert_single</code>	This tells EdgeDB to assume the input has <i>at most</i> one element. This will throw an error if the argument is a set containing more than one element. This is useful if you are changing a property from <code>multi</code> to <code>single</code> .  <code>fill_expr&gt; assert_single(.sheep)</code>
type casts	Useful when converting a property to a different type.  <code>cast_expr&gt; &lt;bigint&gt;.xp</code>

#### 1.6.5.1 Further reading

For guides on advanced migration workflows, refer to the following guides.

- [Making a property required](#)
- [Adding backlinks](#)
- [Changing the type of a property](#)
- [Changing a property to a link](#)
- [Adding a required link](#)

For more information on how migrations work in EdgeDB, check out the [CLI reference](#) or the [Beta 1 blog post](#), which describes the design of the migration system.

## 1.7 EdgeQL

EdgeQL is the query language of EdgeDB. It’s intended as a spiritual successor to SQL that solves some of its biggest design limitations. This page is intended as a rapid-fire overview so you can hit the ground running with EdgeDB. Refer to the linked pages for more in-depth documentation.

Want to follow along with the queries below? Open the [Interactive Tutorial](#) in a separate tab. Copy and paste the queries below and execute them directly from the browser.

---

**Note:** The examples below also demonstrate how to express the query with the [TypeScript client’s](#) query builder, which lets you express arbitrary EdgeQL queries in a code-first, typesafe way.

---

### 1.7.1 Scalar literals

EdgeDB has a rich primitive type system consisting of the following data types.

Strings	<code>str</code>
Booleans	<code>bool</code>
Numbers	<code>int32 int64 float32 float64 bigint decimal</code>
UUID	<code>uuid</code>
JSON	<code>json</code>
Dates and times	<code>datetime cal::local_datetime cal::local_date cal::local_time</code>
Durations	<code>duration cal::relative_duration cal::date_duration</code>
Binary data	<code>bytes</code>
Auto-incrementing counters	<code>sequence</code>
Enums	<code>enum&lt;x, y, z&gt;</code>

Basic literals can be declared using familiar syntax.

Listing 1: edgeql-repl

```
db> select "i edgedb"; # str
{i edgedb'}
db> select false; # bool
{false}
db> select 42; # int64
{42}
db> select 3.14; # float64
{3.14}
db> select 12345678n; # bigint
{12345678n}
db> select 15.0e+100n; # decimal
{15.0e+100n}
db> select b'bina\x01ry'; # bytes
{b'bina\x01ry'}
```

Listing 2: typescript

```
e.str("i edgedb")
// string
e.bool(false)
// boolean
e.int64(42)
// number
e.float64(3.14)
// number
e.bigint(BigInt(12345678))
// bigint
e.decimal("1234.4567")
// n/a (not supported by JS clients)
e.bytes(Buffer.from("bina\x01ry"))
// Buffer
```

Other type literals are declared by *casting* an appropriately structured string.

Listing 3: edgeql-repl

```
db> select <uuid>'a5ea6360-75bd-4c20-b69c-8f317b0d2857';
{a5ea6360-75bd-4c20-b69c-8f317b0d2857}
db> select <datetime>'1999-03-31T15:17:00Z';
{<datetime>'1999-03-31T15:17:00Z'}
db> select <duration>'5 hours 4 minutes 3 seconds';
{<duration>'5:04:03'}
db> select <cal::relative_duration>'2 years 18 days';
{<cal::relative_duration>'P2Y18D'}
```

Listing 4: typescript

```
e.uuid("a5ea6360-75bd-4c20-b69c-8f317b0d2857")
// string
e.datetime("1999-03-31T15:17:00Z")
// Date
e.duration("5 hours 4 minutes 3 seconds")
// edgedb.Duration (custom class)
e.cal.relative_duration("2 years 18 days")
// edgedb.RelativeDuration (custom class)
```

Primitive data can be composed into arrays and tuples, which can themselves be nested.

Listing 5: edgeql-repl

```
db> select ['hello', 'world'];
{['hello', 'world']}
db> select ('Apple', 7, true);
{('Apple', 7, true)} # unnamed tuple
db> select (fruit := 'Apple', quantity := 3.14, fresh := true);
{(fruit := 'Apple', quantity := 3.14, fresh := true)} # unnamed tuple
db> select <json>["this", "is", "an", "array"];
{["this", "is", "an", "array"]}
```

Listing 6: typescript

```
e.array(["hello", "world"]);
// string[]
e.tuple(["Apple", 7, true]);
// [string, number, boolean]
e.tuple({fruit: "Apple", quantity: 3.14, fresh: true});
// {fruit: string; quantity: number; fresh: boolean}
e.json(["this", "is", "an", "array"]);
// unknown
```

EdgeDB also supports a special `json` type for representing unstructured data. Primitive data structures can be converted to JSON using a type cast (`<json>`). Alternatively, a properly JSON-encoded string can be converted to `json` with the built-in `to_json` function. Indexing a `json` value returns another `json` value.

Listing 7: edgeql-repl

```
edgedb> select <json>5;
>{"5"}
edgedb> select <json>[1,2,3];
{[1, 2, 3]}
edgedb> select to_json('[{ "name": "Peter Parker" }]');
{[{"name": "Peter Parker"}]}
edgedb> select to_json('[{ "name": "Peter Parker" }]')[0]['name'];
{"Peter Parker"}
```

Listing 8: typescript

```
/*
  The result of an query returning `json` is represented
  with `unknown` in TypeScript.
*/
e.json(5); // => unknown
e.json([1, 2, 3]); // => unknown
e.to_json('[{ "name": "Peter Parker" }]'); // => unknown
e.to_json('[{ "name": "Peter Parker" }]')[0]["name"]; // => unknown
```

Refer to [Docs > EdgeQL > Literals](#) for complete docs.

## 1.7.2 Functions and operators

EdgeDB provides a rich standard library of functions to operate and manipulate various data types.

Listing 9: edgeql-repl

```
db> select str_upper('oh hi mark');
{'OH HI MARK'}
db> select len('oh hi mark');
{10}
db> select uuid_generate_v1mc();
{c68e3836-0d59-11ed-9379-fb98e50038bb}
db> select contains(['a', 'b', 'c'], 'd');
{false}
```

Listing 10: typescript

```
e.str_upper("oh hi mark");
// string
e.len("oh hi mark");
// number
e.uuid_generate_v1mc();
// string
e.contains(["a", "b", "c"], "d");
// boolean
```

Similarly, it provides a comprehensive set of built-in operators.

Listing 11: edgeql-repl

```
db> select not true;
{false}
db> select exists 'hi';
{true}
db> select 2 + 2;
{4}
db> select 'Hello' ++ ' world!';
{'Hello world!'}
db> select '' if true else '';
{''}
db> select <duration>'5 minutes' + <duration>'2 hours';
{<duration>'2:05:00'}
```

Listing 12: typescript

```
e.op("not", e.bool(true));
// boolean
e.op("exists", e.set("hi"));
// boolean
e.op("exists", e.cast(e.str, e.set()));
// boolean
e.op(e.int64(2), "+", e.int64(2));
// number
e.op(e.str("Hello "), "++", e.str("World!"));
// string
e.op(e.str(""), "if", e.bool(true), "else", e.str(""));
// string
e.op(e.duration("5 minutes"), "+", e.duration("2 hours"))
```

See [Docs > Standard Library](#) for reference documentation on all built-in types, including the functions and operators that apply to them.

### 1.7.3 Insert an object

Objects are created using `insert`. The `insert` statement relies on developer-friendly syntax like curly braces and the `:=` operator.

Listing 13: edgeql

```
insert Movie {
    title := 'Doctor Strange 2',
    release_year := 2022
};
```

Listing 14: typescript

```
const query = e.insert(e.Movie, {
    title: 'Doctor Strange 2',
    release_year: 2022
});
```

(continues on next page)

(continued from previous page)

```
const result = await query.run(client);
// {id: string}
// by default INSERT only returns
// the id of the new object
```

See [Docs > EdgeQL > Insert](#).

#### 1.7.4 Nested inserts

One of EdgeQL's greatest features is that it's easy to compose. Nested inserts are easily achieved with subqueries.

Listing 15: edgedql

```
insert Movie {
    title := 'Doctor Strange 2',
    release_year := 2022,
    director := (insert Person {
        name := 'Sam Raimi'
    })
};
```

Listing 16: typescript

```
const query = e.insert(e.Movie, {
    title: 'Doctor Strange 2',
    release_year: 2022,
    director: e.insert(e.Person, {
        name: 'Sam Raimi'
    })
});

const result = await query.run(client);
// {id: string}
// by default INSERT only returns
// the id of the new object
```

#### 1.7.5 Select objects

Use a *shape* to define which properties to **select** from the given object type.

Listing 17: edgeql

```
select Movie {
    id,
    title
};
```

Listing 18: typescript

```
const query = e.select(e.Movie, () => ({
  id: true,
  title: true
}));
const result = await query.run(client);
// {id: string; title: string}[]

// To select all properties of an object, use the
// spread operator with the special "*" property:
const query = e.select(e.Movie, () => ({
  ...e.Movie['*']
}));
```

Fetch linked objects with a nested shape.

Listing 19: edgeql

```
select Movie {
  id,
  title,
  actors: {
    name
  }
};
```

Listing 20: typescript

```
const query = e.select(e.Movie, () => ({
  id: true,
  title: true,
  actors: {
    name: true,
  }
}));

const result = await query.run(client);
// {id: string; title: string, actors: {name: string}[]}[]
```

See [Docs > EdgeQL > Select > Shapes](#).

### 1.7.6 Filtering, ordering, and pagination

The `select` statement can be augmented with `filter`, `order by`, `offset`, and `limit` clauses (in that order).

Listing 21: edgeql

```
select Movie {
  id,
  title
}
```

(continues on next page)

(continued from previous page)

```
filter .release_year > 2017
order by .title
offset 10
limit 10;
```

Listing 22: typescript

```
const query = e.select(e.Movie, (movie) => ({
    id: true,
    title: true,
    filter: e.op(movie.release_year, ">", 1999),
    order_by: movie.title,
    offset: 10,
    limit: 10,
}));

const result = await query.run(client);
// {id: string; title: number}[]
```

Note that you reference properties of the object to include in your `select` by prepending the property name with a period: `.release_year`. This is known as *leading dot notation*.

Every new set of curly braces introduces a new scope. You can add `filter`, `limit`, and `offset` clauses to nested shapes.

Listing 23: edgeql

```
select Movie {
    title,
    actors: {
        name
    } filter .name ilike 'chris%'
}
filter .title ilike '%avengers%';
```

Listing 24: typescript

```
e.select(e.Movie, movie => ({
  title: true,
  characters: c => ({
    name: true,
    filter: e.op(c.name, "ilike", "chris%"),
  }),
  filter: e.op(movie.title, "ilike", "%avengers%"),
));
// => { characters: { name: string; }[]; title: string; }[]

const result = await query.run(client);
// {id: string; title: number}[]
```

See *Filtering*, *Ordering*, and *Pagination*.

### 1.7.7 Query composition

We've seen how to `insert` and `select`. How do we do both in one query? Answer: query composition. EdgeQL's syntax is designed to be *composable*, like any good programming language.

Listing 25: edgeql

```
select (
  insert Movie { title := 'The Marvels' }
) {
  id,
  title
};
```

Listing 26: typescript

```
const newMovie = e.insert(e.Movie, {
  title: "The Marvels"
});
const query = e.select(newMovie, () => ({
  id: true,
  title: true
}));

const result = await query.run(client);
// {id: string; title: string}
```

We can clean up this query by pulling out the `insert` statement into a `with` block. A `with` block is useful for composing complex multi-step queries, like a script.

Listing 27: edgeql

```
with new_movie := (insert Movie { title := 'The Marvels' })
select new_movie {
  id,
```

(continues on next page)

(continued from previous page)

```
title
};
```

Listing 28: typescript

```
/*
Same as above.

In the query builder, explicit ``with`` blocks aren't necessary!
Just assign your EdgeQL subqueries to variables and compose them as you
like. The query builder automatically convert your top-level query to an
EdgeQL expression with proper ``with`` blocks.
*/
```

## 1.7.8 Computed properties

Selection shapes can contain computed properties.

Listing 29: edgedql

```
select Movie {
    title,
    title_upper := str_upper(.title),
    cast_size := count(.actors)
};
```

Listing 30: typescript

```
e.select(e.Movie, movie => ({
    title: true,
    title_upper: e.str_upper(movie.title),
    cast_size: e.count(movie.actors)
}))
// {title: string; title_upper: string; cast_size: number}[]
```

A common use for computed properties is to query a link in reverse; this is known as a *backlink* and it has special syntax.

Listing 31: edgedql

```
select Person {
    name,
    acted_in := .<actors[is Content] {
        title
    }
};
```

Listing 32: typescript

```
e.select(e.Person, person => ({
    name: true,
```

(continues on next page)

(continued from previous page)

```
acted_in: e.select(person["<actors[is Content]"], () => ({
    title: true,
  })),
});
// {name: string; acted_in: {title: string}[];}[]
```

See [Docs > EdgeQL > Select > Computed](#) and [Docs > EdgeQL > Select > Backlinks](#).

### 1.7.9 Update objects

The update statement accepts a `filter` clause up-front, followed by a `set` shape indicating how the matching objects should be updated.

Listing 33: edgeql

```
update Movie
filter .title = "Doctor Strange 2"
set {
  title := "Doctor Strange in the Multiverse of Madness"
};
```

Listing 34: typescript

```
const query = e.update(e.Movie, (movie) => ({
  filter: e.op(movie.title, '=', 'Doctor Strange 2'),
  set: {
    title: 'Doctor Strange in the Multiverse of Madness',
  },
}));

const result = await query.run(client);
// {id: string}
```

When updating links, the set of linked objects can be added to with `+=`, subtracted from with `-=`, or overwritten with `:=`.

Listing 35: edgeql

```
update Movie
filter .title = "Doctor Strange 2"
set {
  actors += (select Person filter .name = "Rachel McAdams")
};
```

Listing 36: typescript

```
e.update(e.Movie, (movie) => ({
  filter: e.op(movie.title, '=', 'Doctor Strange 2'),
  set: {
    actors: {
      "+=": e.select(e.Person, person => ({
        filter: e.op(person.name, "=", "Rachel McAdams")
      })
    }
  }
});
```

(continues on next page)

(continued from previous page)

```

        }))  
    }  
,  
});
```

See [Docs > EdgeQL > Update](#).

### 1.7.10 Delete objects

The `delete` statement can contain `filter`, `order by`, `offset`, and `limit` clauses.

Listing 37: edgeql

```
delete Movie  
filter .ilike "the avengers%"  
limit 3;
```

Listing 38: typescript

```
const query = e.delete(e.Movie, (movie) => ({  
    filter: e.op(movie.title, 'ilike', "the avengers%"),  
}));  
  
const result = await query.run(client);  
// {id: string}[]
```

See [Docs > EdgeQL > Delete](#).

### 1.7.11 Query parameters

You can reference query parameters in your queries with `$<name>` notation. Since EdgeQL is a strongly typed language, all query parameters must be prepending with a *type cast* to indicate the expected type.

---

**Note:** Scalars like `str`, `int64`, and `json` are supported. Tuples, arrays, and object types are not.

---

Listing 39: edgeql

```
insert Movie {  
    title := <str>$title,  
    release_year := <int64>$release_year  
};
```

Listing 40: typescript

```
const query = e.params({ title: e.str, release_year: e.int64 }, ($) => {  
    return e.insert(e.Movie, {  
        title: $.title,  
        release_year: $.release_year,  
    })
```

(continues on next page)

(continued from previous page)

```
};

const result = await query.run(client, {
    title: 'Thor: Love and Thunder',
    release_year: 2022,
});
// {id: string}
```

All client libraries provide a dedicated API for specifying parameters when executing a query.

Listing 41: javascript

```
import {createClient} from "edgedb";

const client = createClient();
const result = await client.query(`select <str>$param`, {
    param: "Play it, Sam."
});
// => "Play it, Sam."
```

Listing 42: python

```
import edgedb

client = edgedb.create_async_client()

async def main():

    result = await client.query("select <str>$param", param="Play it, Sam")
    # => "Play it, Sam"
```

Listing 43: go

```
package main

import (
    "context"
    "log"

    "github.com/edgedb/edgedb-go"
)

func main() {
    ctx := context.Background()
    client, err := edgedb.CreateClient(ctx, edgedb.Options{})
    if err != nil {
        log.Fatal(err)
    }
    defer client.Close()

    var (
        param     string = "Play it, Sam."
```

(continues on next page)

(continued from previous page)

```

        result  string
    )

    query := "select <str>$0"
    err = client.Query(ctx, query, &result, param)
    // ...
}

```

See [Docs > EdgeQL > Parameters](#).

### 1.7.12 Subqueries

Unlike SQL, EdgeQL is *composable*; queries can be naturally nested. This is useful, for instance, when performing nested mutations.

Listing 44: edgeql

```

with
dr_strange := (select Movie filter .title = "Doctor Strange"),
benedicts := (select Person filter .name in {
    'Benedict Cumberbatch',
    'Benedict Wong'
})
update dr_strange
set {
    actors += benedicts
};

```

Listing 45: typescript

```

// select Doctor Strange
const drStrange = e.select(e.Movie, movie => ({
    filter: e.op(movie.title, '=', "Doctor Strange")
}));

// select actors
const actors = e.select(e.Person, person => ({
    filter: e.op(person.name, 'in', e.set(
        'Benedict Cumberbatch',
        'Benedict Wong'
    ))
}));

// add actors to cast of drStrange
const query = e.update(drStrange, ()=>({
    actors: { "+=": actors }
}));

```

We can also use subqueries to fetch properties of an object we just inserted.

Listing 46: edgeql

```
with new_movie := (insert Movie {
    title := "Avengers: The Kang Dynasty",
    release_year := 2025
})
select new_movie {
    title, release_year
};
```

Listing 47: typescript

```
// "with" blocks are added automatically
// in the generated query!

const newMovie = e.insert(e.Movie, {
    title: "Avengers: The Kang Dynasty",
    release_year: 2025
});

const query = e.select(newMovie, ()=>{
    title: true,
    release_year: true,
});

const result = await query.run(client);
// {title: string; release_year: number;}
```

See [Docs > EdgeQL > Select > Subqueries](#).

### 1.7.13 Polymorphic queries

Consider the following schema.

```
abstract type Content {
    required property title -> str;
}

type Movie extending Content {
    property release_year -> int64;
}

type TVShow extending Content {
    property num_seasons -> int64;
}
```

```
abstract type Content {
    required title: str;
}

type Movie extending Content {
    release_year: int64;
```

(continues on next page)

(continued from previous page)

```

}

type TVShow extending Content {
    num_seasons: int64;
}

```

We can select the abstract type `Content` to simultaneously fetch all objects that extend it, and use the `[is <type>]` syntax to select properties from known subtypes.

Listing 48: edgeql

```

select Content {
    title,
    [is TVShow].num_seasons,
    [is Movie].release_year
};

```

Listing 49: typescript

```

const query = e.select(e.Content, (content) => ({
    title: true,
    ...e.is(e.Movie, {release_year: true}),
    ...e.is(e.TVShow, {num_seasons: true}),
})];
/* {
    title: string;
    release_year: number | null;
    num_seasons: number | null;
}[] */

```

See [Docs > EdgeQL > Select > Polymorphic queries](#).

### 1.7.14 Grouping objects

Unlike SQL, EdgeQL provides a top-level `group` statement to compute groupings of objects.

Listing 50: edgeql

```

group Movie { title, actors: { name } }
by .release_year;

```

Listing 51: typescript

```

e.group(e.Movie, (movie) => {
    const release_year = movie.release_year;
    return {
        title: true,
        by: {release_year},
    };
});
/* {
    grouping: string[];
}

```

(continues on next page)

(continued from previous page)

```
key: { release_year: number | null };
elements: { title: string; }[];
}[] */

```

See [Docs > EdgeQL > Group](#).

## 1.8 Client Libraries

EdgeDB implements libraries for popular languages that make it easier to work with EdgeDB. These libraries provide a common set of functionality.

- *Instantiating clients.* Most libraries implement a `Client` class that internally manages a pool of physical connections to your EdgeDB instance.
- *Resolving connections.* All client libraries implement a standard protocol for determining how to connect to your database. In most cases, this will involve checking for special environment variables like `EDGEDB_DSN`. (More on this in the [Connection](#) section below.)
- *Executing queries.* A `Client` will provide some methods for executing queries against your database. Under the hood, this query is executed using EdgeDB's efficient binary protocol.

---

**Note:** For some use cases, you may not need a client library. EdgeDB allows you to execute [queries over HTTP](#). This is slower than the binary protocol and lacks support for transactions and rich data types, but may be suitable if a client library isn't available for your language of choice.

---

### 1.8.1 Available libraries

To execute queries from your application code, use one of EdgeDB's *client libraries* for the following languages.

- JavaScript/TypeScript
- Go
- Python
- Rust
- .NET

Unofficial (community-maintained) libraries are available for the following languages.

- Elixir

### 1.8.2 Usage

To follow along with the guide below, first create a new directory and initialize a project.

```
$ mydir myproject
$ cd myproject
$ edgedb project init
```

Configure the environment as needed for your preferred language.

Listing 52: Node.js

```
$ npm init -y
$ tsc --init # (TypeScript only)
$ touch index.ts
```

Listing 53: Deno

```
$ touch index.ts
```

Listing 54: Python

```
$ python -m venv venv
$ source venv/bin/activate
$ touch main.py
```

Listing 55: Rust

```
$ cargo init
```

Listing 56: Go

```
$ go mod init example/quickstart
$ touch hello.go
```

Listing 57: .NET

```
$ dotnet new console -o . -f net6.0
```

Install the EdgeDB client library.

Listing 58: Node.js

```
$ npm install edgedb      # npm
$ yarn add edgedb        # yarn
```

Listing 59: Deno

```
n/a
```

Listing 60: Python

```
$ pip install edgedb
```

Listing 61: Rust

```
# Cargo.toml

[dependencies]
edgedb-tokio = "0.3.0"
# additional dependencies
tokio = { version = "1", features = ["full"] }
```

(continues on next page)

(continued from previous page)

```
anyhow = "1.0.63"
```

Listing 62: Go

```
$ go get github.com/edgedb/edgedb-go
```

Listing 63: .NET

```
$ dotnet add package EdgeDB.Net.Driver
```

Copy and paste the following simple script. This script initializes a Client instance. Clients manage an internal pool of connections to your database and provide a set of methods for executing queries.

---

**Note:** Note that we aren't passing connection information (say, a connection URL) when creating a client. The client libraries can detect that they are inside a project directory and connect to the project-linked instance automatically. For details on configuring connections, refer to the [Connection](#) section below.

---

Listing 64: Node.js

```
import {createClient} from 'edgedb';

const client = createClient();

client.querySingle(`select random()`).then((result) => {
  console.log(result);
});
```

Listing 65: Deno

```
import {createClient} from 'https://deno.land/x/edgedb';

const client = createClient();

const result = await client.querySingle(`select random()`);
console.log(result);
```

Listing 66: python

```
from edgedb import create_client

client = create_client()

result = client.query_single("select random()")
print(result)
```

Listing 67: rust

```
// src/main.rs
#[tokio::main]
async fn main() -> anyhow::Result<()> {
```

(continues on next page)

(continued from previous page)

```

let conn = edgedb_tokio::create_client().await?;
let val = conn
    .query_required_single::<f64, _>("select random()", &())
    .await?;
println!("Result: {}", val);
Ok(())
}

```

Listing 68: go

```

// hello.go
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/edgedb/edgedb-go"
)

func main() {
    ctx := context.Background()
    client, err := edgedb.CreateClient(ctx, edgedb.Options{})
    if err != nil {
        log.Fatal(err)
    }
    defer client.Close()

    var result float64
    err = client.
        QuerySingle(ctx, "select random();", &result)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(result)
}

```

Listing 69: .NET

```

using EdgeDB;

var client = new EdgeDBClient();
var result = await client.QuerySingleAsync<double>("select random();");
Console.WriteLine(result);

```

Finally, execute the file.

Listing 70: Node.js

```
$ npx tsx index.ts
```

Listing 71: Deno

```
$ deno run --allow-all --unstable index.deno.ts
```

Listing 72: Python

```
$ python index.py
```

Listing 73: Rust

```
$ cargo run
```

Listing 74: Go

```
$ go run .
```

Listing 75: .NET

```
$ dotnet run
```

You should see a random number get printed to the console. This number was generated inside your EdgeDB instance using EdgeQL's built-in `random()` function.

## 1.8.3 Connection

All client libraries implement a standard protocol for determining how to connect to your database.

### 1.8.3.1 Using projects

In development, we recommend [initializing a project](#) in the root of your codebase.

```
$ edgedb project init
```

Once the project is initialized, any code that uses an official client library will automatically connect to the project-linked instance—no need for environment variables or hard-coded credentials. Follow the [Using projects](#) guide to get started.

### 1.8.3.2 Using EDGEDB\_DSN

In production, connection information can be securely passed to the client library via environment variables. Most commonly, you set a value for `EDGEDB_DSN`.

---

**Note:** If environment variables like `EDGEDB_DSN` are defined inside a project directory, the environment variables will take precedence.

A DSN is also known as a “connection string” and takes the following form.

```
edgedb://<username>:<password>@<hostname>:<port>
```

Each element of the DSN is optional; in fact `edgedb://` is a technically a valid DSN. Any unspecified element will default to the following values.

<host>	localhost
<port>	5656
<user>	edgedb
<password>	null

A typical DSN may look like this:

```
edgedb://username:pas$$word@db.domain.com:8080
```

DSNs can also contain the following query parameters.

<code>database</code>	The database to connect to within the given instance. Defaults to <code>edgedb</code> .
<code>tls_security</code>	The TLS security mode. Accepts the following values. <ul style="list-style-type: none"><li>• "strict" (<b>default</b>) — verify certificates and hostnames</li><li>• "no_host_verification" — verify certificates only</li><li>• "insecure" — trust self-signed certificates</li></ul>
<code>tls_ca_file</code>	A filesystem path pointing to a CA root certificate. This is usually only necessary when attempting to connect via TLS to a remote instance with a self-signed certificate.

These parameters can be added to any DSN using Web-standard query string notation.

```
edgedb://user:pass@example.com:8080?database=my_db&tls_security=insecure
```

For a more comprehensive guide to DSNs, see the [DSN Specification](#).

### 1.8.3.3 Using multiple environment variables

If needed for your deployment pipeline, each element of the DSN can be specified independently.

- `EDGEDB_HOST`
- `EDGEDB_PORT`
- `EDGEDB_USER`
- `EDGEDB_PASSWORD`
- `EDGEDB_DATABASE`
- `EDGEDB_TLS_CA_FILE`
- `EDGEDB_CLIENT_TLS_SECURITY`

---

**Note:** If a value for `EDGEDB_DSN` is defined, it will override these variables!

---

#### 1.8.3.4 Other mechanisms

**EDGEDB\_CREDENTIALS\_FILE** A path to a .json file containing connection information. In some scenarios (including local Docker development) its useful to represent connection information with files.

```
{
  "host": "localhost",
  "port": 10700,
  "user": "testuser",
  "password": "testpassword",
  "database": "edgedb",
  "tls_cert_data": "-----BEGIN CERTIFICATE-----\nabcdef..."
}
```

**EDGEDB\_INSTANCE (local only)** The name of a local instance. Only useful in development.

#### 1.8.3.5 Reference

These are the most common ways to connect to an instance, however EdgeDB supports several other options for advanced use cases. For a complete reference on connection configuration, see [Reference > Connection Parameters](#).

EdgeDB is a next-generation [graph-relational database](#) designed as a spiritual successor to the relational database.

It inherits the strengths of SQL databases: type safety, performance, reliability, and transactionality. But instead of modeling data in a relational (tabular) way, EdgeDB represents data with *object types* containing *properties* and *links* to other objects. It leverages this object-oriented model to provide a superpowered query language that solves some of SQL's biggest usability problems.

## 1.9 How to read the docs

EdgeDB is a complex system, but we've structured the documentation so you can learn it in “phases”. You only need to learn as much as you need to start building your application.

- **Get Started** — Start with the [quickstart](#). It walks through EdgeDB’s core workflows: how to install EdgeDB, create an instance, write a simple schema, execute a migration, write some simple queries, and use the client libraries. The rest of the section goes deeper on each of these subjects.
- **Schema** — A set of pages that break down the concepts of syntax of EdgeDB’s schema definition language (SDL). This starts with a rundown of EdgeDB’s primitive type system ([Primitives](#)), followed by a description of ([Object Types](#)) and the things they can contain: links, properties, indexes, access policies, and more.
- **EdgeQL** — A set of pages that break down EdgeDB’s query language, EdgeQL. It starts with a rundown of how to declare [literal values](#), then introduces some key EdgeQL concepts like sets, paths, and type casts. With the basics established, it proceeds to break down all of EdgeQL’s top-level statements: `select`, `insert`, and so on.
- **Guides** — Contains collections of guides on topics that are peripheral to EdgeDB itself: how to deploy to various cloud providers, how to integrate with various frameworks, and how to introspect the schema to build code-generation tools on top of EdgeDB.
- **Standard Library** — This section contains an encyclopedic breakdown of EdgeDB’s built-in types and the functions/operators that can be used with them. We didn’t want to clutter the **EdgeQL** section with all the nitty-gritty on each of these. If you’re looking for a particular function (say, a `replace`), go to the Standard Library page for the relevant type (in this case, [String](#)), and peruse the table for what you’re looking for (`str_replace()`).

- **Client Libraries** The documentation for EdgeDB’s set of official client libraries for JavaScript/TypeScript, Python, Go, and Rust. All client libraries implement EdgeDB’s binary protocol and provide a standard interface for executing queries. If you’re using another language, you can execute queries [over HTTP](#). This section also includes documentation for EdgeDB’s [GraphQL](#) endpoint.
- **CLI** Complete reference for the `edgedb` command-line tool. The CLI is self-documenting—add the `--help` flag after any command to print the relevant documentation—so you shouldn’t need to reference this section often.
- **Reference** The *Reference* section contains a complete breakdown of EdgeDB’s *syntax* (for both EdgeQL and SDL), *internals* (like the binary protocol and dump file format), and *configuration settings*. Usually you’ll only need to reference these once you’re an advanced user.
- **Changelog** Detailed changelogs for each successive version of EdgeDB, including any breaking changes, new features, bigfixes, and links to

## 1.10 Tooling

To actually build apps with EdgeDB, you’ll need to know more than SDL and EdgeQL.

- **CLI** — The most commonly used CLI functionality is covered in the [Quickstart](#). For additional details, we have dedicated guides for [Migrations](#) and [Projects](#). A full CLI reference is available under [CLI](#).
- **Client Libraries** — To actually execute queries, you’ll use one of our client libraries for JavaScript, Go, or Python; find your preferred library under [Client Libraries](#). If you’re using another language, you can still use EdgeDB! You can execute [queries via HTTP](#).
- **Deployment** — To publish an EdgeDB-backed application, you’ll need to deploy EdgeDB. Refer to [Guides > Deployment](#) for step-by-step deployment guides for all major cloud hosting platforms, as well as instructions for self-hosting with Docker.

EdgeDB features:

- strict, strongly typed schema;
- powerful and clean query language;
- ability to easily work with complex hierarchical data;
- built-in support for schema migrations.

EdgeDB is not a graph database: the data is stored and queried using relational database techniques. Unlike most graph databases, EdgeDB maintains a strict schema.

EdgeDB is not a document database, but inserting and querying hierarchical document-like data is trivial.

EdgeDB is not a traditional object database, despite the classification, it is not an implementation of OOP persistence.

## 2.1 Primitives

EdgeDB has a robust type system consisting of primitive and object types. Below is a review of EdgeDB's primitive types; later, these will be used to declare *properties* on object types.

### 2.1.1 Scalar types

<code>str</code>	A variable-length string
<code>bool</code>	Logical boolean (true/false)
<code>int16</code>	16-bit integer
<code>int32</code>	32-bit integer
<code>int64</code>	64-bit integer
<code>float32</code>	32-bit floating point number
<code>float64</code>	64-bit floating point number
<code>bigint</code>	Arbitrary precision integer
<code>decimal</code>	Arbitrary precision number
<code>json</code>	Arbitrary JSON data
<code>uuid</code>	UUID type
<code>bytes</code>	Raw binary data
<code>datetime</code>	Timezone-aware point in time
<code>duration</code>	Absolute time span
<code>cal::local_datetime</code>	Date and time without timezone
<code>cal::local_date</code>	Date type
<code>cal::local_time</code>	Time type
<code>cal::relative_duration</code>	Relative time span (in months, days, and seconds)
<code>cal::date_duration</code>	Relative time span (in months and days only)
<code>sequence</code>	Auto-incrementing sequence of <code>int64</code>

Custom scalar types can also be declared. For full documentation, see [SDL > Scalar types](#).

## 2.1.2 Enums

To represent an enum, declare a custom scalar that extends the abstract `enum` type.

```
scalar type Color extending enum<Red, Green, Blue>;  
  
type Shirt {  
    property color -> Color;  
}
```

```
scalar type Color extending enum<Red, Green, Blue>;  
  
type Shirt {  
    color: Color;  
}
```

---

**Important:** To reference enum values inside EdgeQL queries, use dot notation, e.g. `Color.Green`.

---

For a full reference on enum types, see the [Enum docs](#).

## 2.1.3 Arrays

Arrays store zero or more primitive values of the same type in an ordered list. Arrays cannot contain object types or other arrays.

```
type Person {  
    property str_array -> array<str>;  
    property json_array -> array<json>;  
  
    # INVALID: arrays of object types not allowed  
    # property friends -> array<Person>  
  
    # INVALID: arrays cannot be nested  
    # property nested_array -> array<array<str>>  
}
```

```
type Person {  
    str_array: array<str>;  
    json_array: array<json>;  
  
    # INVALID: arrays of object types not allowed  
    # property friends -> array<Person>  
  
    # INVALID: arrays cannot be nested  
    # property nested_array -> array<array<str>>  
}
```

For a full reference on array types, see the [Array docs](#).

## 2.1.4 Tuples

Like arrays, tuples are ordered sequences of primitive data. Unlike arrays, each element of a tuple can have a distinct type. Tuple elements can be *any type*, including primitives, objects, arrays, and other tuples.

```
type Person {

    property unnamed_tuple -> tuple<str, bool, int64>;
    property nested_tuple -> tuple<tuple<str, tuple<bool, int64>>>;
    property tuple_of_arrays -> tuple<array<str>, array<int64>>;

}
```

```
type Person {

    unnamed_tuple: tuple<str, bool, int64>;
    nested_tuple: tuple<tuple<str, tuple<bool, int64>>>;
    tuple_of_arrays: tuple<array<str>, array<int64>>;

}
```

Optionally, you can assign a *key* to each element of the tuple. Tuples containing explicit keys are known as *named tuples*. You must assign keys to all elements (or none of them).

```
type BlogPost {
    property metadata -> tuple<title: str, published: bool, upvotes: int64>;
}
```

```
type BlogPost {
    metadata: tuple<title: str, published: bool, upvotes: int64>;
}
```

Named and unnamed tuples are the same data structure under the hood. You can add, remove, and change keys in a tuple type after it's been declared. For details, see [EdgeQL > Literals > Tuples](#).

---

**Important:** When you query an *unnamed* tuple using one of EdgeQL's *client libraries*, its value is converted to a list/array. When you fetch a named tuple, it is converted into an object/dictionary/hashmap depending on the language.

## 2.1.5 Ranges

Ranges represent some interval of values. The intervals can be bound or unbound on either end. They can also be empty, containing no values. Only some scalar types have corresponding range types:

- range<int32>
- range<int64>
- range<float32>
- range<float64>
- range<decimal>
- range<datetime>

- `range<cal::local_datetime>`
- `range<cal::local_date>`

```
type DieRoll {  
    property values -> range<int64>;  
}
```

```
type DieRoll {  
    values: range<int64>;  
}
```

For a full reference on ranges, functions and operators see the [Range docs](#).

## 2.1.6 Sequences

To represent an auto-incrementing integer property, declare a custom scalar that extends the abstract `sequence` type. Creating a sequence type initializes a global `int64` counter that auto-increments whenever a new object is created. All properties that point to the same sequence type will share the counter.

```
scalar type ticket_number extending sequence;  
type Ticket {  
    property number -> ticket_number;  
}
```

```
scalar type ticket_number extending sequence;  
type Ticket {  
    number: ticket_number;  
}
```

For a full reference on sequences, see the [Sequence docs](#).

## 2.2 Object Types

*Object types* are the primary components of an EdgeDB schema. They are analogous to SQL *tables* or ORM *models*, and consist of *properties* and *links*.

Properties are used to attach primitive data to an object type. They are declared with the `property` keyword. For the full documentation on properties, see [Properties](#).

```
type Person {  
    property email -> str;  
}
```

```
type Person {  
    email: str;  
}
```

Links are used to define relationships between object types. Prior to EdgeDB 3.0, they must be declared with the `link` keyword, but the keyword is not required in 3.0+ unless the link is computed (e.g., backlinks). For the full documentation on links, see [Links](#).

```
type Person {
    link best_friend -> Person;
}
```

```
type Person {
    best_friend: Person;
}
```

## 2.2.1 IDs

There's no need to manually declare a primary key on your object types. All object types automatically contain a property `id` of type `UUID` that's *required*, *globally unique*, and *readonly*. This `id` is assigned upon creation and never changes.

## 2.2.2 Abstract types

Object types can either be *abstract* or *non-abstract*. By default all object types are non-abstract. You can't create or store instances of abstract types, but they're a useful way to share functionality and structure among other object types.

```
abstract type HasName {
    property first_name -> str;
    property last_name -> str;
}
```

```
abstract type HasName {
    first_name: str;
    last_name: str;
}
```

Abstract types are commonly used in tandem with inheritance.

## 2.2.3 Inheritance

Object types can *extend* other object types. The extending type (AKA the *subtype*) inherits all links, properties, indexes, constraints, etc. from its *supertypes*.

```
abstract type Animal {
    property species -> str;
}
```

```
type Dog extending Animal {
    property breed -> str;
}
```

```
abstract type Animal {
    species: str;
}
```

```
type Dog extending Animal {
```

(continues on next page)

(continued from previous page)

```
breed: str;
}
```

### 2.2.3.1 Multiple Inheritance

Object types can *extend more than one type* — that's called *multiple inheritance*. This mechanism allows building complex object types out of combinations of more basic types.

```
abstract type HasName {
    property first_name -> str;
    property last_name -> str;
}

abstract type HasEmail {
    property email -> str;
}

type Person extending HasName, HasEmail {
    property profession -> str;
}
```

```
abstract type HasName {
    first_name: str;
    last_name: str;
}

abstract type HasEmail {
    email: str;
}

type Person extending HasName, HasEmail {
    profession: str;
}
```

If multiple supertypes share links or properties, those properties must be of the same type and cardinality.

---

**Note:** Refer to the dedicated pages on [Indexes](#), [Constraints](#), [Object-Level Security](#), and [Annotations](#) for documentation on these concepts.

---

See also
<a href="#">SDL &gt; Object types</a>
<a href="#">DDL &gt; Object types</a>
<a href="#">Introspection &gt; Object types</a>
<a href="#">Cheatsheets &gt; Object types</a>

## 2.3 Properties

### index property

Properties are used to associate primitive data with an [object type](#) or [link](#).

```
type Player {
    property email -> str;
    property points -> int64;
    property is_online -> bool;
}
```

```
type Player {
    email: str;
    points: int64;
    is_online: bool;
}
```

Properties are associated with a *key* (e.g. `first_name`) and a primitive type (e.g. `str`). The term *primitive type* is an umbrella term that encompasses [scalar types](#) like `str` and `bool`, [enums](#), [arrays](#), and [tuples](#).

### 2.3.1 Required properties

Properties can be either optional (the default) or required.

```
type User {
    required property email -> str;
}
```

```
type User {
    required email: str;
}
```

### 2.3.2 Property cardinality

Properties have a **cardinality**, either `single` (the default) or `multi`. A `multi` property of type `str` points to an *unordered set* of strings.

```
type User {

    # single isn't necessary here
    # properties are single by default
    single property name -> str;

    # an unordered set of strings
    multi property nicknames -> str;

    # an unordered set of string arrays
    multi property set_of_arrays -> array<str>;
}
```

```
type User {  
  
    # single isn't necessary here  
    # properties are single by default  
    single name: str;  
  
    # an unordered set of strings  
    multi nicknames: str;  
  
    # an unordered set of string arrays  
    multi set_of_arrays: array<str>;  
}
```

### Comparison to arrays

The values associated with a `multi` property are stored in no particular order. If order is important, use an `array`. Otherwise, `multi` properties are recommended. For a more involved discussion, see [EdgeQL > Sets](#).

### 2.3.3 Default values

Properties can have a default value. This default can be a static value or an arbitrary EdgeQL expression, which will be evaluated upon insertion.

```
type Player {  
    required property points -> int64 {  
        default := 0;  
    }  
  
    required property latitude -> float64 {  
        default := (360 * random() - 180);  
    }  
}
```

```
type Player {  
    required points: int64 {  
        default := 0;  
    }  
  
    required latitude: float64 {  
        default := (360 * random() - 180);  
    }  
}
```

### 2.3.4 Readonly properties

Properties can be marked as `readonly`. In the example below, the `User.external_id` property can be set at the time of creation but not modified thereafter.

```
type User {
    required property external_id -> uuid {
        readonly := true;
    }
}
```

```
type User {
    required external_id: uuid {
        readonly := true;
    }
}
```

### 2.3.5 Constraints

Properties can be augmented with constraints. The example below showcases a subset of EdgeDB's built-in constraints.

```
type BlogPost {
    property title -> str {
        constraint exclusive; # all post titles must be unique
        constraint min_len_value(8);
        constraint max_len_value(30);
        constraint regexp(r'^[A-Za-z0-9 ]+$');
    }

    property status -> str {
        constraint one_of('Draft', 'InReview', 'Published');
    }

    property upvotes -> int64 {
        constraint min_value(0);
        constraint max_value(9999);
    }
}
```

```
type BlogPost {
    title: str {
        constraint exclusive; # all post titles must be unique
        constraint min_len_value(8);
        constraint max_len_value(30);
        constraint regexp(r'^[A-Za-z0-9 ]+$');
    }

    status: str {
        constraint one_of('Draft', 'InReview', 'Published');
    }

    upvotes: int64 {
```

(continues on next page)

(continued from previous page)

```

    constraint min_value(0);
    constraint max_value(9999);
}
}

```

You can constrain properties with arbitrary [EdgeQL](#) expressions returning `bool`. To reference the value of the property, use the special scope keyword `__subject__`.

```

type BlogPost {
    property title -> str {
        constraint expression on (
            __subject__ = str_trim(__subject__)
        );
    }
}

```

```

type BlogPost {
    title: str {
        constraint expression on (
            __subject__ = str_trim(__subject__)
        );
    }
}

```

The constraint above guarantees that `BlogPost.title` doesn't contain any leading or trailing whitespace by checking that the raw string is equal to the trimmed version. It uses the built-in `str_trim()` function.

For a full reference of built-in constraints, see the [Constraints reference](#).

### 2.3.6 Annotations

Properties can contain annotations, small human-readable notes. The built-in annotations are `title`, `description`, and `deprecated`. You may also declare [custom annotation types](#).

```

type User {
    property email -> str {
        annotation title := 'Email address';
        annotation description := "The user's email address.";
        annotation deprecated := 'Use NewUser instead.';
    }
}

```

```

type User {
    email: str {
        annotation title := 'Email address';
        annotation description := "The user's email address.";
        annotation deprecated := 'Use NewUser instead.';
    }
}

```

### 2.3.7 Abstract properties

Properties can be *concrete* (the default) or *abstract*. Abstract properties are declared independent of a source or target, can contain *annotations*, and can be marked as `readonly`.

```
abstract property email_prop {
    annotation title := 'An email address';
    readonly := true;
}

type Student {
    # inherits annotations and "readonly := true"
    property email extending email_prop -> str;
}
```

```
abstract property email_prop {
    annotation title := 'An email address';
    readonly := true;
}

type Student {
    # inherits annotations and "readonly := true"
    email: str {
        extending email_prop;
    };
}
```

### 2.3.8 Link properties

Properties can also be defined on **links**. For a full guide, refer to [Guides > Using link properties](#).

See also
<a href="#">SDL &gt; Properties</a>
<a href="#">DDL &gt; Properties</a>
<a href="#">Introspection &gt; Object types</a>

## 2.4 Links

**index** link one-to-one one-to-many many-to-one many-to-many

Links define a specific relationship between two *object types*.

### 2.4.1 Defining links

```
type Person {  
    link best_friend -> Person;  
}
```

```
type Person {  
    best_friend: Person;  
}
```

Links are *directional*; they have a source (the object type on which they are declared) and a *target* (the type they point to).

### 2.4.2 Link cardinality

All links have a cardinality: either **single** or **multi**. The default is **single** (a “to-one” link). Use the **multi** keyword to declare a “to-many” link.

```
type Person {  
    multi link friends -> Person;  
}
```

```
type Person {  
    multi friends: Person;  
}
```

On the other hand, backlinks work in reverse to find objects that link to the object, and thus assume **multi** as a default. Use the **single** keyword to declare a “to-one” backlink.

```
type Author {  
    link posts := .<authors[is Article];  
}  
  
type CompanyEmployee {  
    single link company := .<employees[is Company];  
}
```

### 2.4.3 Required links

All links are either `optional` or `required`; the default is `optional`. Use the `required` keyword to declare a required link. A required link must point to *at least one* target instance. In this scenario, every Person must have a `best_friend`:

```
type Person {
    required link best_friend -> Person;
}
```

```
type Person {
    required best_friend: Person;
}
```

Links with cardinality `multi` can also be `required`; `required multi` links must point to *at least one* target object.

```
type Person {
    property name -> str;
}

type GroupChat {
    required multi link members -> Person;
}
```

```
type Person {
    name: str;
}

type GroupChat {
    required multi members: Person;
}
```

In this scenario, each `GroupChat` must contain at least one person. Attempting to create a `GroupChat` with no members would fail.

### 2.4.4 Exclusive constraints

You can add an exclusive constraint to a link to guarantee that no other instances can link to the same target(s).

```
type Person {
    property name -> str;
}

type GroupChat {
    required multi link members -> Person {
        constraint exclusive;
    }
}
```

```
type Person {
    name: str;
}
```

(continues on next page)

(continued from previous page)

```
type GroupChat {
    required multi members: Person {
        constraint exclusive;
    }
}
```

In the `GroupChat` example, the `GroupChat.members` link is now `exclusive`. Two `GroupChat` objects cannot link to the same `Person`; put differently, no `Person` can be a member of multiple `GroupChat`.

## 2.4.5 Modeling relations

By combining *link cardinality* and *exclusivity constraints*, we can model every kind of relationship: one-to-one, one-to-many, many-to-one, and many-to-many.

Relation type	Cardinality	Exclusive
One-to-one	<code>single</code>	Yes
One-to-many	<code>multi</code>	Yes
Many-to-one	<code>single</code>	No
Many-to-many	<code>multi</code>	No

### 2.4.5.1 Many-to-one

Many-to-one relationships typically represent concepts like ownership, membership, or hierarchies. For example, `Person` and `Shirt`. One person may own many shirts, and a shirt is (usually) owned by just one person.

```
type Person {
    required property name -> str
}

type Shirt {
    required property color -> str;
    link owner -> Person;
}
```

```
type Person {
    required name: str
}

type Shirt {
    required color: str;
    owner: Person;
}
```

Since links are `single` by default, each `Shirt` only corresponds to one `Person`. In the absence of any exclusivity constraints, multiple shirts can link to the same `Person`. Thus, we have a one-to-many relationship between `Person` and `Shirt`.

When fetching a `Person`, it's possible to deeply fetch their collection of `Shirts` by traversing the `Shirt.owner` link *in reverse*. This is known as a **backlink**; read the [select docs](#) to learn more.

### 2.4.5.2 One-to-many

Conceptually, one-to-many and many-to-one relationships are identical; the “directionality” of a relation is just a matter of perspective. Here, the same “shirt owner” relationship is represented with a `multi` link.

```
type Person {
    required property name -> str;
    multi link shirts -> Shirt {
        # ensures a one-to-many relationship
        constraint exclusive;
    }
}

type Shirt {
    required property color -> str;
}
```

```
type Person {
    required name: str;
    multi shirts: Shirt {
        # ensures a one-to-many relationship
        constraint exclusive;
    }
}

type Shirt {
    required color: str;
}
```

---

**Note:** Don’t forget the exclusive constraint! This is required to ensure that each `Shirt` corresponds to a single `Person`. Without it, the relationship will be many-to-many.

---

Under the hood, a `multi` link is stored in an intermediate `association table`, whereas a `single` link is stored as a column in the object type where it is declared. As a result, single links are marginally more efficient. Generally `single` links are recommended when modeling 1:N relations.

### 2.4.5.3 One-to-one

Under a *one-to-one* relationship, the source object links to a single instance of the target type, and vice versa. As an example consider a schema to represent assigned parking spaces.

```
type Employee {
    required property name -> str;
    link assigned_space -> ParkingSpace {
        constraint exclusive;
    }
}

type ParkingSpace {
    required property number -> int64;
}
```

```
type Employee {
    required name: str;
    assigned_space: ParkingSpace {
        constraint exclusive;
    }
}

type ParkingSpace {
    required number: int64;
}
```

All links are `single` unless otherwise specified, so no `Employee` can have more than one `assigned_space`. Moreover, the `exclusive` constraint guarantees that a given `ParkingSpace` can't be assigned to multiple employees at once. Together the `single` link and exclusivity constraint constitute a *one-to-one* relationship.

#### 2.4.5.4 Many-to-many

A *many-to-many* relation is the least constrained kind of relationship. There is no exclusivity or cardinality constraints in either direction. As an example consider a simple app where a `User` can “like” their favorite `Movies`.

```
type User {
    required property name -> str;
    multi link likes -> Movie;
}

type Movie {
    required property title -> str;
}
```

```
type User {
    required name: str;
    multi likes: Movie;
}

type Movie {
    required title: str;
}
```

A user can like multiple movies. And in the absence of an `exclusive` constraint, each movie can be liked by multiple users. Thus this is a *many-to-many* relationship.

#### 2.4.6 Default values

Like properties, links can declare a default value in the form of an EdgeQL expression, which will be executed upon insertion. In the example below, new people are automatically assigned three random friends.

```
type Person {
    required property name -> str;
    multi link friends -> Person {
        default := (select Person order by random() limit 3);
    }
}
```

```
type Person {
    required name: str;
    multi friends: Person {
        default := (select Person order by random() limit 3);
    }
}
```

## 2.4.7 Link properties

Like object types, links in EdgeDB can contain **properties**. Link properties can be used to store metadata about links, such as *when* they were created or the *nature/strength* of the relationship.

```
type Person {
    property name -> str;
    multi link family_members -> Person {
        property relationship -> str;
    }
}
```

```
type Person {
    name: str;
    multi family_members: Person {
        relationship: str;
    }
}
```

Above, we model a family tree with a single `Person` type. The `Person. family_members` link is a many-to-many relation; each `family_members` link can contain a string `relationship` describing the relationship of the two individuals.

Due to how they're persisted under the hood, link properties must always be `single` and `optional`.

In practice, link properties are most useful with many-to-many relationships. In that situation there's a significant difference between the *relationship* described by the link and the *target object*. Thus it makes sense to separate properties of the relationships and properties of the target objects. On the other hand, for one-to-one, one-to-many, and many-to-one relationships there's an exact correspondence between the link and one of the objects being linked. In these situations any property of the relationship can be equally expressed as the property of the target object (for one-to-many and one-to-one cases) or as the property of the source object (for many-to-one and one-to-one cases). It is generally advisable to use object properties instead of link properties in these cases due to better ergonomics of selecting, updating, and even casting into `json` when keeping all data in the same place rather than spreading it across link and object properties.

---

**Note:** For a full guide on modeling, inserting, updating, and querying link properties, see the [Using Link Properties](#) guide.

## 2.4.8 Deletion policies

Links can declare their own **deletion policy**. There are two kinds of events that might trigger these policies: *target deletion* and *source deletion*.

### 2.4.8.1 Target deletion

Target deletion policies determine what action should be taken when the *target* of a given link is deleted. They are declared with the `on target delete` clause.

```
type MessageThread {
    property title -> str;
}

type Message {
    property content -> str;
    link chat -> MessageThread {
        on target delete delete source;
    }
}
```

```
type MessageThread {
    title: str;
}

type Message {
    content: str;
    chat: MessageThread {
        on target delete delete source;
    }
}
```

The `Message.chat` link in the example uses the `delete source` policy. There are 4 available target deletion policies.

- `restrict` (default) - Any attempt to delete the target object immediately raises an exception.
- `delete source` - when the target of a link is deleted, the source is also deleted. This is useful for implementing cascading deletes.

---

**Note:** There is a [limit](#) to the depth of a deletion cascade due to an upstream stack size limitation.

---

- `allow` - the target object is deleted and is removed from the set of the link targets.
- `deferred restrict` - any attempt to delete the target object raises an exception at the end of the transaction, unless by that time this object is no longer in the set of link targets.

### 2.4.8.2 Source deletion

Source deletion policies determine what action should be taken when the *source* of a given link is deleted. They are declared with the `on source delete` clause.

```
type MessageThread {
    property title -> str;
    multi link messages -> Message {
        on source delete delete target;
    }
}

type Message {
    property content -> str;
}
```

```
type MessageThread {
    title: str;
    multi messages: Message {
        on source delete delete target;
    }
}

type Message {
    content: str;
}
```

Under this policy, deleting a `MessageThread` will *unconditionally* delete its `messages` as well.

To avoid deleting a `Message` that is linked to by other `MessageThread` objects via their `message` link, append `if orphan` to that link's deletion policy.

```
type MessageThread {
    property title -> str;
    multi link messages -> Message {
-        on source delete delete target;
+        on source delete delete target if orphan;
    }
}
```

```
type MessageThread {
    title: str;
    multi messages: Message {
-        on source delete delete target;
+        on source delete delete target if orphan;
    }
}
```

---

**Note:** Deletion policies using `if orphan` will result in the target being deleted *unless it is linked by another object via the same link the policy is on*. This qualifier does not apply globally across all links in the database or across different links even if they're on the same type. For example, a `Message` might be linked from both a `MessageThread` and a `Channel1`. If the `MessageThread` linking to it is deleted, the deletion policy would still result in the `Message` being deleted as long as no other `MessageThread` objects link to it on that same field. It is orphaned with respect to the

MessageThread type's link field, even though it is not orphaned globally.

Similarly, if the MessageThread had two links both linking to messages — maybe the existing messages link and another called related to link other related Message objects that are not in the thread — if orphan could result in linked messages being deleted even if they were also linked from another MessageThread object's related link because they were orphaned with respect to the messages link.

---

## 2.4.9 Polymorphic links

Links can have abstract targets, in which case the link is considered **polymorphic**. Consider the following schema:

```
abstract type Person {
    property name -> str;
}

type Hero extending Person {
    # additional fields
}

type Villain extending Person {
    # additional fields
}
```

```
abstract type Person {
    name: str;
}

type Hero extending Person {
    # additional fields
}

type Villain extending Person {
    # additional fields
}
```

The abstract type Person has two concrete subtypes: Hero and Villain. Despite being abstract, Person can be used as a link target in concrete object types.

```
type Movie {
    property title -> str;
    multi link characters -> Person;
}
```

```
type Movie {
    title: str;
    multi characters: Person;
}
```

In practice, the Movie.characters link can point to a Hero, Villain, or any other non-abstract subtype of Person. For details on how to write queries on such a link, refer to the [Polymorphic Queries docs](#)

## 2.4.10 Abstract links

It's possible to define abstract links that aren't tied to a particular *source* or *target*. If you're declaring several links with the same set of properties, annotations, constraints, or indexes, abstract links can be used to eliminate repetitive SDL.

```
abstract link link_with_strength {
    property strength -> float64;
    index on (__subject__@strength);
}

type Person {
    multi link friends extending link_with_strength -> Person;
}
```

```
abstract link link_with_strength {
    strength: float64;
    index on (__subject__@strength);
}

type Person {
    multi friends: Person {
        extending link_with_strength;
    };
}
```

See also
<a href="#">SDL &gt; Links</a>
<a href="#">DDL &gt; Links</a>
<a href="#">Introspection &gt; Object types</a>

## 2.5 Computed

**edb-alt-title** Computed properties and links

---

**Important:** This section assumes a basic understanding of EdgeQL. If you aren't familiar with it, feel free to skip this page for now.

---

Object types can contain *computed* links and properties. Computed properties and links are not persisted in the database. Instead, they are evaluated *on the fly* whenever that field is queried.

```
type Person {
    property name -> str;
    property all_caps_name := str_upper(__subject__.name);
}
```

```
type Person {
    name: str;
    property all_caps_name := str_upper(__subject__.name);
}
```

Computed fields are associated with an EdgeQL expression. This expression can be an *arbitrary* EdgeQL query. This expression is evaluated whenever the field is referenced in a query.

---

**Note:** Computed fields don't need to be pre-defined in your schema; you can drop them into individual queries as well. They behave in exactly the same way. For more information, see the [EdgeQL > Select > Computeds](#).

---

**Warning:** Volatile functions are not allowed in computed properties defined in schema. This means that, for example, your schema-defined computed property cannot call `datetime_current()`, but it *can* call `datetime_of_transaction()` or `datetime_of_statement()`. This does *not* apply to computed properties outside of schema.

### 2.5.1 Leading dot notation

The example above used the special keyword `__subject__` to refer to the current object; it's analogous to `this` or `self` in many object-oriented languages.

However, explicitly using `__subject__` is optional here; inside the scope of an object type declaration, you can omit it entirely and use the `.<name>` shorthand.

```
type Person {
    property first_name -> str;
    property last_name -> str;
    property full_name := .first_name ++ ' ' ++ .last_name;
}
```

```
type Person {
    first_name: str;
    last_name: str;
    property full_name := .first_name ++ ' ' ++ .last_name;
}
```

### 2.5.2 Type and cardinality inference

The type and cardinality of a computed field is *inferred* from the expression. There's no need for the modifier keywords you use for non-computed fields (like `multi` and `required`). However, it's common to specify them anyway; it makes the schema more readable and acts as a sanity check: if the provided EdgeQL expression disagrees with the modifiers, an error will be thrown the next time you try to *create a migration*.

```
type Person {
    property first_name -> str;

    # this is invalid, because first_name is not a required property
    required property first_name_upper := str_upper(.first_name);
}
```

```
type Person {
    first_name: str;
```

(continues on next page)

(continued from previous page)

```
# this is invalid, because first_name is not a required property
required property first_name_upper := str_upper(.first_name);
}
```

## 2.5.3 Common use cases

### 2.5.3.1 Filtering

If you find yourself writing the same filter expression repeatedly in queries, consider defining a computed field that encapsulates the filter.

```
type Club {
    multi link members -> Person;
    multi link active_members := (
        select .members filter .is_active = true
    )
}

type Person {
    property name -> str;
    property is_active -> bool;
}
```

```
type Club {
    multi members: Person;
    multi link active_members := (
        select .members filter .is_active = true
    )
}

type Person {
    name: str;
    is_active: bool;
}
```

### 2.5.3.2 Backlinks

Backlinks are one of the most common use cases for computed links. In EdgeDB links are *directional*; they have a source and a target. Often it's convenient to traverse a link in the *reverse* direction.

```
type BlogPost {
    property title -> str;
    link author -> User;
}

type User {
    property name -> str;
    multi link blog_posts := .<author[is BlogPost]
}
```

```
type BlogPost {
    title: str;
    author: User;
}

type User {
    name: str;
    multi link blog_posts := .<author[is BlogPost]
}
```

The `User.blog_posts` expression above uses the *backlink operator* `.<` in conjunction with a *type filter* [`is BlogPost`] to fetch all the `BlogPosts` associated with a given `User`. For details on this syntax, see the EdgeQL docs for [Backlinks](#).

### 2.5.3.3 Created Timestamp

Using a computed property, you can timestamp when an object was created in your database.

```
type BlogPost {
    property title -> str;
    link author -> User;
    required property created_at -> datetime {
        readonly := true;
        default := datetime_of_statement();
    }
}
```

```
type BlogPost {
    title: str;
    author: User;
    required created_at: datetime {
        readonly := true;
        default := datetime_of_statement();
    }
}
```

When a `BlogPost` is created, `datetime_of_statement()` will be called to supply it with a timestamp as the `created_at` property. You might also consider `datetime_of_transaction()` if that's better suited to your use case.

<a href="#">SDL &gt; Links</a>
<a href="#">DDL &gt; Links</a>
<a href="#">SDL &gt; Properties</a>
<a href="#">DDL &gt; Properties</a>

## 2.6 Indexes

An index is a data structure used internally by a database to speed up filtering and sorting operations. Most commonly, indexes are declared within object type declarations and reference a particular property; this will speed up any query that references that property in a `filter` or `order by` clause.

---

**Note:** While improving query performance, indexes also increase disk and memory usage and slow down insertions and updates. Creating too many indexes may be detrimental; only index properties you often filter or order by.

---

### 2.6.1 Index on a property

Below, we are referencing the `User.name` property with the *dot notation shorthand*: `.name`.

```
type User {
    required property name -> str;
    index on (.name);
}
```

```
type User {
    required name: str;
    index on (.name);
}
```

By indexing on `User.name`, queries that filter by the `name` property will be faster, as the database can look up a name in the index instead of scanning through all Users sequentially.

### 2.6.2 Index on an expression

Indexes may be defined using an arbitrary *singleton* expression that references multiple properties of the enclosing object type.

---

**Important:** A singleton expression is an expression that's guaranteed to return *at most one* element. As such, you can't index on a `multi` property.

---

```
type User {
    required property first_name -> str;
    required property last_name -> str;
    index on (str_lower(.firstname + ' ' + .lastname));
}
```

```
type User {
    required first_name: str;
    required last_name: str;
    index on (str_lower(.firstname + ' ' + .lastname));
}
```

### 2.6.3 Index on multiple properties

A *composite index* is an index that references multiple properties. This will speed up queries that filter or sort on *both properties*. In EdgeDB, this is accomplished by indexing on a `tuple` of properties.

```
type User {
    required property name -> str;
    required property email -> str;
    index on ((.name, .email));
}
```

```
type User {
    required name: str;
    required email: str;
    index on ((.name, .email));
}
```

### 2.6.4 Index on a link property

Link properties can also be indexed.

```
abstract link friendship {
    property strength -> float64;
    index on (__subject__@strength);
}

type User {
    multi link friends extending friendship -> User;
}
```

```
abstract link friendship {
    strength: float64;
    index on (__subject__@strength);
}

type User {
    multi friends: User {
        extending friendship;
    };
}
```

### 2.6.5 Specify a Postgres index type

EdgeDB exposes Postgres indexes that you can use in your schemas. These are exposed through the `pg` module.

- `pg::hash`- Index based on a 32-bit hash derived from the indexed value
- `pg::btree`- B-tree index can be used to retrieve data in sorted order
- `pg::gin`- GIN is an “inverted index” appropriate for data values that contain multiple elements, such as arrays and JSON
- `pg::gist`- GIST index can be used to optimize searches involving ranges

- `pg::spgist`- SP-GIST index can be used to optimize searches involving ranges and strings
- `pg::brin`- BRIN (Block Range INdex) index works with summaries about the values stored in consecutive physical block ranges in the database

You can use them like this:

```
type User {
    required property name -> str;
    index pg::spgist on (.name);
};
```

## 2.6.6 Annotate an index

Indexes can be augmented with annotations.

```
type User {
    property name -> str;
    index on (.name) {
        annotation description := 'Indexing all users by name.';
    };
}
```

```
type User {
    name: str;
    index on (.name) {
        annotation description := 'Indexing all users by name.';
    };
}
```

---

### Important: Foreign and primary keys

In SQL databases, indexes are commonly used to index *primary keys* and *foreign keys*. In EdgeDB, these fields are automatically indexed; there's no need to manually declare them. Moreover, any property with an `exclusive` constraint is also automatically indexed.

See also
<a href="#">SDL &gt; Indexes</a>
<a href="#">DDL &gt; Indexes</a>
<a href="#">Introspection &gt; Indexes</a>

## 2.7 Constraints

---

**Important:** This section assumes a basic understanding of EdgeQL.

---

Constraints give users fine-grained control over which data is considered valid. They can be defined on `properties`, `links`, `object types`, and `custom scalars`.

Below is a simple property constraint.

```
type User {
    required property username -> str {
        constraint exclusive;
    }
}
```

```
type User {
    required username: str {
        constraint exclusive;
    }
}
```

This example uses a built-in constraint, `exclusive`. Refer to the table below for a complete list; click the name of a given constraint for the full documentation.

<code>exclusive</code>	Enforce uniqueness among all instances of the containing type
<code>expression</code>	Custom constraint expression
<code>one_of</code>	A list of allowable values
<code>max_value</code>	Maximum value numerically/lexicographically
<code>max_ex_value</code>	Maximum value numerically/lexicographically (exclusive range)
<code>max_len_value</code>	Maximum length (strings only)
<code>min_value</code>	Minimum value numerically/lexicographically
<code>min_ex_value</code>	Minimum value numerically/lexicographically (exclusive range)
<code>min_len_value</code>	Minimum length (strings only)
<code>regexp</code>	Regex constraint (strings only)

## 2.7.1 Constraints on properties

The `max_len_value` constraint below uses the built-in `len()` function, which returns the length of a string.

```
type User {
    required property username -> str {
        # usernames must be unique
        constraint exclusive;

        # max length (built-in)
        constraint max_len_value(25);
    };
}
```

```
type User {
    required username: str {
        # usernames must be unique
        constraint exclusive;

        # max length (built-in)
        constraint max_len_value(25);
    };
}
```

### 2.7.1.1 Custom constraints

The `expression` constraint is used to define custom constraint logic. Inside custom constraints, the keyword `__subject__` can be used to reference the *value* being constrained.

```
type User {
    required property username -> str {
        # max length (as custom constraint)
        constraint expression on (len(__subject__) <= 25);
    };
}
```

```
type User {
    required username: str {
        # max length (as custom constraint)
        constraint expression on (len(__subject__) <= 25);
    };
}
```

### 2.7.2 Constraints on object types

Constraints can be defined on object types. This is useful when the constraint logic must reference multiple links or properties.

**Important:** Inside an object type declaration, you can omit `__subject__` and simply refer to properties with the *leading dot notation* (e.g. `.<name>`).

```
type ConstrainedVector {
    required property x -> float64;
    required property y -> float64;

    constraint expression on (
        .x ^ 2 + .y ^ 2 <= 25
    );
}
```

```
type ConstrainedVector {
    required x: float64;
    required y: float64;

    constraint expression on (
        .x ^ 2 + .y ^ 2 <= 25
    );
}
```

Note that the constraint expression cannot contain arbitrary EdgeQL! Due to how constraints are implemented, you can only reference **single** (non-multi) properties and links defined on the object type.

```
# Not valid!
type User {
```

(continues on next page)

(continued from previous page)

```

required property username -> str;
multi link friends -> User;

# constraints cannot contain paths with more than one hop
constraint expression on ('bob' in .friends.username);
}

```

```

# Not valid!
type User {
    required username: str;
    multi friends: User;

    # constraints cannot contain paths with more than one hop
    constraint expression on ('bob' in .friends.username);
}

```

### 2.7.2.1 Computed constraints

Constraints can be defined on computed properties.

```

type User {
    required property username -> str;
    required property clean_username := str_trim(str_lower(.username));

    constraint exclusive on (.clean_username);
}

```

```

type User {
    required username: str;
    required property clean_username := str_trim(str_lower(.username));

    constraint exclusive on (.clean_username);
}

```

### 2.7.2.2 Composite constraints

To define a composite constraint, create an **exclusive** constraint on a tuple of properties or links.

```

type User {
    property username -> str;
}

type BlogPost {
    property title -> str;
    link author -> User;

    constraint exclusive on ((.title, .author));
}

```

```

type User {
    username: str;
}

type BlogPost {
    title: str;
    author: User;

    constraint exclusive on ((.title, .author));
}

```

### 2.7.2.3 Partial constraints

Constraints on object types can be made partial, so that they don't apply when some condition holds.

```

type User {
    required property username -> str;
    property deleted -> bool;

    # Usernames must be unique unless marked deleted
    constraint exclusive on (.username) except (.deleted);
}

```

```

type User {
    required username: str;
    deleted: bool;

    # Usernames must be unique unless marked deleted
    constraint exclusive on (.username) except (.deleted);
}

```

### 2.7.3 Constraints on links

When defining a constraint on a link, `__subject__` refers to the *link itself*. This is commonly used add constraints to *link properties*.

```

type User {
    property name -> str;
    multi link friends -> User {
        single property strength -> float64;
        constraint expression on (
            __subject__@strength >= 0
        );
    }
}

```

```

type User {
    name: str;
    multi friends: User {
        single strength: float64;
}

```

(continues on next page)

(continued from previous page)

```
constraint expression on (
    __subject__@strength >= 0
);
}
```

## 2.7.4 Constraints on custom scalars

Custom scalar types can be constrained.

```
scalar type username extending str {
    constraint regexp(r'^[A-Za-z0-9]{4,20}$');
}
```

Note: you can't use `exclusive` constraints on custom scalar types, as the concept of exclusivity is only defined in the context of a given object type.

Use `expression` constraints to declare custom constraints using arbitrary EdgeQL expressions. The example below uses the built-in `str_trim()` function.

```
scalar type title extending str {
    constraint expression on (
        __subject__ = str_trim(__subject__)
    );
}
```

See also
<a href="#">SDL &gt; Constraints</a>
<a href="#">DDL &gt; Constraints</a>
<a href="#">Introspection &gt; Constraints</a>
<a href="#">Standard Library &gt; Constraints</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; Constraints</a>

## 2.8 Aliases

**Important:** This section assumes a basic understanding of EdgeQL. If you aren't familiar with it, feel free to skip this page for now.

An **alias** is a *pointer* to a set of values. This set is defined with an arbitrary EdgeQL expression.

Like computed properties, this expression is evaluated on the fly whenever the alias is referenced in a query. Unlike computed properties, aliases are defined independent of an object type; they are standalone expressions.

### Scalar alias

```
alias digits := {0,1,2,3,4,5,6,7,8,9};
```

### Object type alias

The name of a given object type (e.g. `User`) is itself a pointer to the *set of all User objects*. After declaring the alias below, you can use `User` and `UserAlias` interchangably.

```
alias UserAlias := User;
```

### Object type alias with computeds

Object type aliases can include a *shape* that declare additional computed properties or links.

```
type Post {
    required property title -> str;
}

alias PostAlias := Post {
    trimmed_title := str_trim(.title)
}
```

```
type Post {
    required title: str;
}

alias PostAlias := Post {
    trimmed_title := str_trim(.title)
}
```

In effect, this creates a *virtual subtype* of the base type, which can be referenced in queries just like any other type.

### Query alias

Aliases can correspond to an arbitrary EdgeQL expression, including entire queries.

```
type BlogPost {
    required property title -> str;
    required property is_published -> bool;
}

alias PublishedPosts := (
    select BlogPost
    filter .is_published = true
);
```

```
type BlogPost {
    required title: str;
    required is_published: bool;
}

alias PublishedPosts := (
    select BlogPost
    filter .is_published = true
);
```

---

**Note:** All aliases are reflected in the database's built-in *GraphQL schema*.

---

See also
<a href="#">SDL &gt; Aliases</a>
<a href="#">DDL &gt; Aliases</a>
<a href="#">Cheatsheets &gt; Aliases</a>

## 2.9 Annotations

*Annotations* are named values associated with schema items and are designed to hold arbitrary schema-level metadata represented as a `str`.

### 2.9.1 Standard annotations

There are a number of annotations defined in the standard library. The following are the annotations which can be set on any schema item:

- `title`
- `description`
- `deprecated`

For example, consider the following declaration:

```
type Status {  
    annotation title := 'Activity status';  
    annotation description := 'All possible user activities';  
  
    required property name -> str {  
        constraint exclusive  
    }  
}
```

```
type Status {  
    annotation title := 'Activity status';  
    annotation description := 'All possible user activities';  
  
    required name: str {  
        constraint exclusive  
    }  
}
```

The `deprecated` annotation is used to mark deprecated items (e.g. `str_rpad()`) and to provide some information such as what should be used instead.

## 2.9.2 User-defined annotations

To declare a custom constraint type beyond the three built-ins, add an abstract annotation type to your schema. A custom annotation could be used to attach arbitrary JSON-encoded data to your schema—potentially useful for introspection and code generation.

```
abstract annotation admin_note;

type Status {
    annotation admin_note := 'system-critical';
}
```

See also
<a href="#">SDL &gt; Annotations</a>
<a href="#">DDL &gt; Annotations</a>
<a href="#">Cheatsheets &gt; Annotations</a>
<a href="#">Introspection &gt; Object types</a>

## 2.10 Globals

Schemas can contain scalar-typed *global variables*.

```
global current_user_id -> uuid;
```

```
global current_user_id: uuid;
```

These provide a useful mechanism for specifying session-level data that can be referenced in queries with the `global` keyword.

```
select User {
    id,
    posts: { title, content }
}
filter .id = global current_user_id;
```

As in the example above, this is particularly useful for representing the notion of a session or “current user”.

### 2.10.1 Setting global variables

Global variables are set when initializing a client. The exact API depends on which client library you’re using.

Listing 1: typescript

```
import createClient from 'edgedb';

const baseClient = createClient()
const clientWithGlobals = baseClient.withGlobals({
    current_user_id: '2141a5b4-5634-4ccc-b835-437863534c51',
});

await clientWithGlobals.query(`select global current_user_id;`);
```

Listing 2: python

```
from edgedb import create_client

client = create_client().with_globals({
    'current_user_id': '580cc652-8ab8-4a20-8db9-4c79a4b1fd81'
})

result = client.query("""
    select global current_user_id;
""")
print(result)
```

Listing 3: go

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/edgedb/edgedb-go"
)

func main() {
    ctx := context.Background()
    client, err := edgedb.CreateClient(ctx, edgedb.Options{})
    if err != nil {
        log.Fatal(err)
    }
    defer client.Close()

    id, err := edgedb.ParseUUID("2141a5b4-5634-4ccc-b835-437863534c51")
    if err != nil {
        log.Fatal(err)
    }

    var result edgedb.UUID
    err = client.
        WithGlobals(map[string]interface{}{"current_user": id}).
        QuerySingle(ctx, "SELECT global current_user;", &result)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(result)
}
```

Listing 4: edgeql

```
set global current_user_id := <uuid>'2141a5b4-5634-4ccc-b835-437863534c51';
```

The `.withGlobals/.with_globals` method returns a new `Client` instance that stores the provided globals and sends them along with all future queries.

### 2.10.1.1 Cardinality

Global variables can be marked `required`; in this case, you must specify a default value.

```
required global one_string -> str {
    default := "Hi Mom!"
};
```

```
required global one_string: str {
    default := "Hi Mom!"
};
```

### 2.10.1.2 Computed globals

Global variables can also be computed. The value of computed globals are dynamically computed when they are referenced in queries.

```
required global random_global := datetime_of_transaction();
```

The provided expression will be computed at the start of each query in which the global is referenced. There's no need to provide an explicit type; the type is inferred from the computed expression.

Computed globals are not subject to the same constraints as non-computed ones; specifically, they can be object-typed and have a `multi` cardinality.

```
global current_user_id -> uuid;

# object-typed global
global current_user := (
    select User filter .id = global current_user_id
);

# multi global
global current_user_friends := (global current_user).friends;
```

```
global current_user_id: uuid;

# object-typed global
global current_user := (
    select User filter .id = global current_user_id
);

# multi global
global current_user_friends := (global current_user).friends;
```

### 2.10.1.3 Usage in schema

Unlike query parameters, globals can be referenced *inside your schema declarations*.

```
type User {
    property name -> str;
    property is_self := (.id = global current_user_id)
};
```

```
type User {
    name: str;
    property is_self := (.id = global current_user_id)
};
```

This is particularly useful when declaring [access policies](#).

```
type Person {
    required property name -> str;
    access policy my_policy allow all using (.id = global current_user_id);
}
```

```
type Person {
    required name: str;
    access policy my_policy allow all using (.id = global current_user_id);
}
```

Refer to [Access Policies](#) for complete documentation.

See also
<a href="#">SDL &gt; Globals</a>
<a href="#">DDL &gt; Globals</a>

## 2.11 Access Policies

Object types can contain security policies that restrict the set of objects that can be selected, inserted, updated, or deleted by a particular query. This is known as *object-level security*.

Let's start with a simple schema without any access policies.

```
type User {
    required property email -> str { constraint exclusive; };
}

type BlogPost {
    required property title -> str;
    required link author -> User;
}
```

```
type User {
    required email: str { constraint exclusive; };
}
```

(continues on next page)

(continued from previous page)

```
type BlogPost {
    required title: str;
    required author: User;
}
```

When no access policies are defined, object-level security is not activated. Any properly authenticated client can select or modify any object in the database.

**Warning:** Once a policy is added to a particular object type, **all operations** (select, insert, delete, update, etc.) on any object of that type are now *disallowed by default* unless specifically allowed by an access policy!

### 2.11.1 Defining a global

To start, we'll add a *global variable* to our schema. We'll use this global to represent the identity of the user executing the query.

```
+ global current_user -> uuid;

type User {
    required property email -> str { constraint exclusive; };
}

type BlogPost {
    required property title -> str;
    required link author -> User;
}
```

```
+ global current_user: uuid;

type User {
    required email: str { constraint exclusive; };
}

type BlogPost {
    required title: str;
    required author: User;
}
```

Global variables are a generic mechanism for providing *context* to a query. Most commonly, they are used in the context of access policies.

The value of these variables is attached to the *client* you use to execute queries. The exact API depends on which client library you're using:

Listing 5: typescript

```
import createClient from 'edgedb';

const client = createClient().withGlobals({
```

(continues on next page)

(continued from previous page)

```
current_user: '2141a5b4-5634-4ccc-b835-437863534c51',
});

await client.query(`select global current_user;`);
```

Listing 6: python

```
from edgedb import create_client

client = create_client().with_globals({
    'current_user': '580cc652-8ab8-4a20-8db9-4c79a4b1fd81'
})

result = client.query("""
    select global current_user;
""")
```

Listing 7: go

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/edgedb/edgedb-go"
)

func main() {
    ctx := context.Background()
    client, err := edgedb.CreateClient(ctx, edgedb.Options{})
    if err != nil {
        log.Fatal(err)
    }
    defer client.Close()

    id, err := edgedb.ParseUUID("2141a5b4-5634-4ccc-b835-437863534c51")
    if err != nil {
        log.Fatal(err)
    }

    var result edgedb.UUID
    err = client.
        WithGlobals(map[string]interface{}{"current_user": id}).
        QuerySingle(ctx, "SELECT global current_user;", &result)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println(result)
}
```

## 2.11.2 Defining a policy

Let's add a policy to our sample schema.

```
global current_user -> uuid;

type User {
    required property email -> str { constraint exclusive; };
}

type BlogPost {
    required property title -> str;
    required link author -> User;

+   access policy author_has_full_access
+       allow all
+       using (global current_user ?= .author.id);
}
```

```
global current_user: uuid;

type User {
    required email: str { constraint exclusive; };
}

type BlogPost {
    required title: str;
    required author: User;

+   access policy author_has_full_access
+       allow all
+       using (global current_user ?= .author.id);
}
```

Let's break down the access policy syntax piece-by-piece. This policy grants full read-write access (`all`) to the author of each `BlogPost`. No one else will be able to edit, delete, or view this post.

---

**Note:** We're using the *coalescing equality* operator `?=` which returns `false` even if one of its arguments is an empty set.

---

- `access policy`: The keyword used to declare a policy inside an object type.
- `author_has_full_access`: The name of this policy; could be any string.
- `allow`: The kind of policy; could be `allow` or `deny`
- `all`: The set of operations being allowed/denied; a comma-separated list of the following: `all, select, insert, delete, update, update read, update write`.
- `using (<expr>)`: A boolean expression. Think of this as a `filter` expression that defines the set of objects to which the policy applies.

Let's do some experiments.

```
db> insert User { email := "test@edgedb.com" };
{default::User {id: be44b326-03db-11ed-b346-7f1594474966}}
db> set global current_user := <uuid>"be44b326-03db-11ed-b346-7f1594474966";
OK: SET GLOBAL
db> insert BlogPost {
...     title := "My post",
...     author := (select User filter .id = global current_user)
... };
{default::BlogPost {id: e76afeae-03db-11ed-b346-fbb81f537ca6}}
```

We've created a `User`, set the value of `current_user` to its `id`, and created a new `BlogPost`. When we try to select all `BlogPost` objects, we'll see the post we just created.

```
db> select BlogPost;
{default::BlogPost {id: e76afeae-03db-11ed-b346-fbb81f537ca6}}
db> select count(BlogPost);
{1}
```

Now let's unset `current_user` and see what happens.

```
db> set global current_user := {};
OK: SET GLOBAL
db> select BlogPost;
{}
db> select count(BlogPost);
{0}
```

Now `select BlogPost` returns zero results. We can only `select` the *posts* written by the *user* specified by `current_user`. When `current_user` has no value, we can't read any posts.

The access policies use global variables to define a “subgraph” of data that is visible to a particular query.

### 2.11.3 Policy types

For the most part, the policy types correspond to EdgeQL's *statement types*:

- `select`: Applies to all queries; objects without a `select` permission cannot be modified either.
- `insert`: Applies to insert queries; executed *post-insert*. If an inserted object violates the policy, the query will fail.
- `delete`: Applies to delete queries.
- `update`: Applies to update queries.

Additionally, the `update` operation can be broken down into two sub-policies: `update read` and `update write`.

- `update read`: This policy restricts *which* objects can be updated. It runs *pre-update*; that is, this policy is executed before the updates have been applied.
- `update write`: This policy restricts *how* you update the objects; you can think of it as a *post-update* validity check. This could be used to prevent a `User` from transferring a `BlogPost` to another `User`.

Finally, there's an umbrella policy that can be used as a shorthand for all the others.

- `all`: A shorthand policy that can be used to allow or deny full read/ write permissions. Exactly equivalent to `select`, `insert`, `update`, `delete`.

## 2.11.4 Resolution order

An object type can contain an arbitrary number of access policies, including several conflicting `allow` and `deny` policies. EdgeDB uses a particular algorithm for resolving these policies.

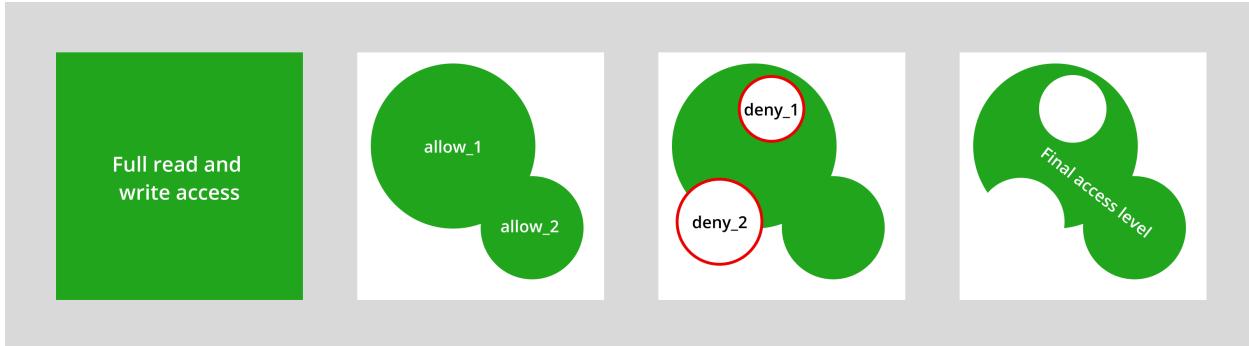


Fig. 1: The access policy resolution algorithm, explained with Venn diagrams.

1. When no policies are defined on a given object type, all objects of that type can be read or modified by any appropriately authenticated connection.
2. EdgeDB then applies all `allow` policies. Each policy grants a *permission* that is scoped to a particular *set of objects* as defined by the `using` clause. Conceptually, these permissions are merged with the `union` / `or` operator to determine the set of allowable actions.
3. After the `allow` policies are resolved, the `deny` policies can be used to carve out exceptions to the `allow` rules. Deny rules *supersede* allow rules! As before, the set of objects targeted by the policy is defined by the `using` clause.
4. This results in the final access level: a set of objects targetable by each of `select`, `insert`, `update`, `read`, `update`, `write`, and `delete`.

Currently, by default the access policies affect the values visible in expressions of *other* access policies. This means that they can affect each other in various ways. Because of this, great care needs to be taken when creating access policies based on objects other than the ones they are defined on. For example:

```
global current_user_id -> uuid;
global current_user := (
    select User filter .id = global current_user_id
);

type User {
    required property email -> str { constraint exclusive; };
    required property is_admin -> bool { default := false };

    access policy admin_only
        allow all
        using (global current_user.is_admin ?? false);
}

type BlogPost {
    required property title -> str;
    link author -> User;

    access policy author_has_full_access
```

(continues on next page)

(continued from previous page)

```

allow all
using (global current_user ?= .author.id);
}

global current_user_id: uuid;
global current_user := (
    select User filter .id = global current_user_id
);

type User {
    required email: str { constraint exclusive; };
    required is_admin: bool { default := false };

    access policy admin_only
        allow all
        using (global current_user.is_admin ?? false);
}

type BlogPost {
    required title: str;
    author: User;

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
}

```

In the above schema only the admin will see a non-empty `author` link, because only the admin can see any user objects at all. This means that instead of making `BlogPost` visible to its author, all non-admin authors won't be able to see their own posts. The above issue can be remedied by making the current user able to see their own `User` record.

**Note:** Starting with EdgeDB 3.0, access policy restrictions will **not** apply to any access policy expression. This means that when reasoning about access policies it is no longer necessary to take other policies into account. Instead, all data is visible for the purpose of *defining* an access policy.

This change is being made to simplify reasoning about access policies and to allow certain patterns to be express efficiently. Since those who have access to modifying the schema can remove unwanted access policies, no additional security is provided by applying access policies to each other's expressions.

It is possible (and recommended) to enable this *future* behavior in EdgeDB 2.6 and later by adding the following to the schema: `using future nonrecursive_access_policies;`

## 2.11.5 Custom error messages

When you run a query that attempts a write and is restricted by an access policy, you will get a generic error message.

```
edgedb error: AccessPolicyError: access policy violation on insert of
<type>
```

---

**Note:** When attempting a `select` queries, you simply won't get the data that is being restricted by the access policy.

---

If you have multiple access policies, it can be useful to know which policy is restricting your query and provide a friendly error message. You can do this by adding a custom error message to your policy.

```
global current_user_id -> uuid;
global current_user := (
    select User filter .id = global current_user_id
);

type User {
    required property email -> str { constraint exclusive; };
    required property is_admin -> bool { default := false };

    access policy admin_only
        allow all
+       using (global current_user.is_admin ?? false) {
+           errmessage := 'Only admins may query Users'
+       };
    }

type BlogPost {
    required property title -> str;
    link author -> User;

    access policy author_has_full_access
        allow all
+       using (global current_user ?= .author.id) {
+           errmessage := 'BlogPosts may only be queried by their authors'
+       };
    }
```

Now if you attempt, for example, a `User` insert as a non-admin user, you will receive this error:

```
edgedb error: AccessPolicyError: access policy violation on insert of
default::User (Only admins may query Users)
```

## 2.11.6 Disabling policies

You may disable all access policies by setting the `apply_access_policies` *configuration parameter* to `false`.

You may also toggle access policies using the “Disable Access Policies” checkbox in the “Config” dropdown in the EdgeDB UI (accessible by running the CLI command `edgedb ui` from inside your project). This is the most convenient way to temporarily disable access policies since it applies only to your UI session.

## 2.11.7 Examples

Blog posts are publicly visible if published but only writable by the author.

```
global current_user -> uuid;

type User {
    required property email -> str { constraint exclusive; };
}

type BlogPost {
    required property title -> str;
    required link author -> User;
+   required property published -> bool { default := false }

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy visible_if_published
+       allow select
+       using (.published);
}
```

```
global current_user: uuid;

type User {
    required email: str { constraint exclusive; };
}

type BlogPost {
    required title: str;
    required author: User;
+   required published: bool { default := false }

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy visible_if_published
+       allow select
+       using (.published);
}
```

Blog posts are visible to friends but only modifiable by the author.

```

global current_user -> uuid;

type User {
    required property email -> str { constraint exclusive; };
+   multi link friends -> User;
}

type BlogPost {
    required property title -> str;
    required link author -> User;

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy friends_can_read
+     allow select
+     using ((global current_user in .author.friends.id) ?? false);
}

```

```

global current_user: uuid;

type User {
    required email: str { constraint exclusive; };
+   multi friends: User;
}

type BlogPost {
    required title: str;
    required author: User;

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy friends_can_read
+     allow select
+     using ((global current_user in .author.friends.id) ?? false);
}

```

Blog posts are publicly visible except to users that have been blocked by the author.

```

type User {
    required property email -> str { constraint exclusive; };
+   multi link blocked -> User;
}

type BlogPost {
    required property title -> str;
    required link author -> User;

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy anyone_can_read
}

```

(continues on next page)

(continued from previous page)

```
+     allow select;
+     access policy exclude_blocked
+     deny select
+     using ((global current_user in .author.blocked.id) ?? false);
}
```

```
type User {
    required email: str { constraint exclusive; };
+   multi blocked: User;
}

type BlogPost {
    required title: str;
    required author: User;

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy anyone_can_read
+     allow select;
+   access policy exclude_blocked
+     deny select
+     using ((global current_user in .author.blocked.id) ?? false);
}
```

“Disappearing” posts that become invisible after 24 hours.

```
type User {
    required property email -> str { constraint exclusive; };
}

type BlogPost {
    required property title -> str;
    required link author -> User;
+   required property created_at -> datetime {
+     default := datetime_of_statement() # non-volatile
+   }

    access policy author_has_full_access
        allow all
        using (global current_user ?= .author.id);
+   access policy hide_after_24hrs
+     allow select
+     using (datetime_of_statement() - .created_at < <duration>'24 hours');
}
```

```
type User {
    required email: str { constraint exclusive; };
}

type BlogPost {
    required title: str;
```

(continues on next page)

(continued from previous page)

```

required author: User;
+ required created_at: datetime {
+   default := datetime_of_statement() # non-volatile
+ }

access policy author_has_full_access
  allow all
  using (global current_user ?= .author.id);
+ access policy hide_after_24hrs
+   allow select
+   using (datetime_of_statement() - .created_at < <duration>'24 hours');
}

```

### 2.11.7.1 Super constraints

Access policies support arbitrary EdgeQL and can be used to define “super constraints”. Policies on `insert` and `update` `write` can be thought of as post-write “validity checks”; if the check fails, the write will be rolled back.

---

**Note:** Due to an underlying Postgres limitation, *constraints on object types* can only reference properties, not links.

---

Here’s a policy that limits the number of blog posts a `User` can post.

```

type User {
  required property email -> str { constraint exclusive; };
+  multi link posts := .<author[is BlogPost]
}

type BlogPost {
  required property title -> str;
  required link author -> User;

  access policy author_has_full_access
    allow all
    using (global current_user ?= .author.id);
+  access policy max_posts_limit
+    deny insert
+    using (count(.author.posts) > 500);
}

```

```

type User {
  required email: str { constraint exclusive; };
+  multi link posts := .<author[is BlogPost]
}

type BlogPost {
  required title: str;
  required author: User;

  access policy author_has_full_access
    allow all
}

```

(continues on next page)

(continued from previous page)

```
    using (global current_user ?= .author.id);
+ access policy max_posts_limit
+   deny insert
+   using (count(.author.posts) > 500);
}
```

<b>See also</b>
<a href="#">SDL &gt; Access policies</a>
<a href="#">DDL &gt; Access policies</a>

## 2.12 Functions

---

**Note:** This page documents how to define custom functions, however EdgeDB provides a large library of built-in functions and operators. These are documented in [Standard Library](#).

---

Functions are ways to transform one set of data into another.

### 2.12.1 User-defined Functions

It is also possible to define custom functions. For example, consider a function that adds an exclamation mark '!' at the end of the string:

```
function exclamation(word: str) -> str
  using (word ++ '!');
```

This function accepts a `str` as an argument and produces a `str` as output as well.

```
test> select exclamation({'Hello', 'World'});
{'Hello!', 'World!}'
```

<b>See also</b>
<a href="#">SDL &gt; Functions</a>
<a href="#">DDL &gt; Functions</a>
<a href="#">Reference &gt; Function calls</a>
<a href="#">Introspection &gt; Functions</a>
<a href="#">Cheatsheets &gt; Functions</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; User-Defined Functions</a>

## 2.13 Triggers

Triggers allow you to define an expression to be executed whenever a given query type is run on an object type. The original query will *trigger* your pre-defined expression to run in a transaction along with the original query. These can be defined in your schema.

---

**Note:** Triggers cannot be used to modify the object that set off the trigger. This functionality will be addressed by the upcoming [mutation rewrites](#) feature.

---

Here's an example that creates a simple audit log type so that we can keep track of what's happening to our users in a database. First, we will create a Log type:

```
type Log {
    action: str;
    timestamp: datetime {
        default := datetime_current();
    }
    target_name: str;
    change: str;
}
```

With the Log type in place, we can write some triggers that will automatically create Log objects for any insert, update, or delete queries on the Person type:

```
type Person {
    required name: str;

    trigger log_insert after insert for each do (
        insert Log {
            action := 'insert',
            target_name := __new__.name
        }
    );

    trigger log_update after update for each do (
        insert Log {
            action := 'update',
            target_name := __new__.name,
            change := __old__.name ++ '->' ++ __new__.name
        }
    );

    trigger log_delete after delete for each do (
        insert Log {
            action := 'delete',
            target_name := __old__.name
        }
    );
}
```

In a trigger's expression, we have access to the `__old__` and/or `__new__` variables which capture the object before and after the query. Triggers on update can use both variables. Triggers on delete can use `__old__`. Triggers on insert can use `__new__`.

---

**Note:** If you access an object in the trigger expression `_without_` using `__old__` or `__new__` (for example, via a `select` query for that object), you will get the object state `_after_` the triggering query. This is the same data you will get if the object is part of the query and you access it via `__new__`.

---

Now, whenever we run a query, we get a log entry as well:

```
db> insert Person {name := 'Jonathan Harker'};
{default::Person {id: b4d4e7e6-bd19-11ed-8363-1737d8d4c3c3}}
db> select Log {action, timestamp, target_name, change};
{
  default::Log {
    action: 'insert',
    timestamp: <datetime>'2023-03-07T18:56:02.403817Z',
    target_name: 'Jonathan Harker',
    change: {}
  }
}
db> update Person filter .name = 'Jonathan Harker'
... set {name := 'Mina Murray'};
{default::Person {id: b4d4e7e6-bd19-11ed-8363-1737d8d4c3c3}}
db> select Log {action, timestamp, target_name, change};
{
  default::Log {
    action: 'insert',
    timestamp: <datetime>'2023-03-07T18:56:02.403817Z',
    target_name: 'Jonathan Harker',
    change: {}
  },
  default::Log {
    action: 'update',
    timestamp: <datetime>'2023-03-07T18:56:39.520889Z',
    target_name: 'Mina Murray',
    change: 'Jonathan Harker->Mina Murray'
  },
}
db> delete Person filter .name = 'Mina Murray';
{default::Person {id: b4d4e7e6-bd19-11ed-8363-1737d8d4c3c3}}
db> select Log {action, timestamp, target_name, change};
{
  default::Log {
    action: 'insert',
    timestamp: <datetime>'2023-03-07T18:56:02.403817Z',
    target_name: 'Jonathan Harker',
    change: {}
  },
  default::Log {
    action: 'update',
    timestamp: <datetime>'2023-03-07T18:56:39.520889Z',
    target_name: 'Mina Murray',
    change: 'Jonathan Harker->Mina Murray'
  },
  default::Log {
```

(continues on next page)

(continued from previous page)

```

action: 'delete',
timestamp: <datetime>'2023-03-07T19:00:52.636084Z',
target_name: 'Mina Murray',
change: {}
},
}

```

**Note:** In some cases, a trigger can cause another trigger to fire. When this happens, EdgeDB completes all the triggers fired by the initial query before kicking off a new “stage” of triggers. In the second stage, any triggers fired by the initial stage of triggers will fire. EdgeDB will continue adding trigger stages until all triggers are complete.

The exception to this is when triggers would cause a loop or would cause the same trigger to be run in two different stages. These triggers will generate an error.

You might find that one log entry per row is too granular or too noisy for your use case. In that case, a `for all` trigger may be a better fit. Here’s a schema that changes the `Log` type so that each object can log multiple writes by making `target_name` and `change` *multi properties* and switches to `for all` triggers:

```

type Log {
    action: str;
    timestamp: datetime {
        default := datetime_current();
    }
-   target_name: str;
-   change: str;
+   multi target_name: str;
+   multi change: str;
}

type Person {
    required name: str;

-   trigger log_insert after insert for each do (
+   trigger log_insert after insert for all do (
        insert Log {
            action := 'insert',
            target_name := __new__.name
        }
    );
}

-   trigger log_update after update for each do (
+   trigger log_update after update for all do (
        insert Log {
            action := 'update',
            target_name := __new__.name,
            change := __old__.name ++ '->' ++ __new__.name
        }
    );
}

-   trigger log_delete after delete for each do (
+   trigger log_delete after delete for all do (

```

(continues on next page)

(continued from previous page)

```

insert Log {
    action := 'delete',
    target_name := __old__.name
}
);
}

```

Under this new schema, each query matching the trigger gets a single Log object instead of one Log object per row:

```

db> for name in {'Jonathan Harker', 'Mina Murray', 'Dracula'}
... union (
...   insert Person {name := name}
... );
{
  default::Person {id: 3836f9c8-d393-11ed-9638-3793d3a39133},
  default::Person {id: 38370a8a-d393-11ed-9638-d3e9b92ca408},
  default::Person {id: 38370abc-d393-11ed-9638-5390f3cbd375},
}
db> select Log {action, timestamp, target_name, change};
{
  default::Log {
    action: 'insert',
    timestamp: <datetime>'2023-03-07T19:12:21.113521Z',
    target_name: {'Jonathan Harker', 'Mina Murray', 'Dracula'},
    change: {},
  },
}
db> for change in {
...   (old_name := 'Jonathan Harker', new_name := 'Jonathan'),
...   (old_name := 'Mina Murray', new_name := 'Mina')
... }
... union (
...   update Person filter .name = change.old_name set {
...     name := change.new_name
...   }
... );
{
  default::Person {id: 3836f9c8-d393-11ed-9638-3793d3a39133},
  default::Person {id: 38370a8a-d393-11ed-9638-d3e9b92ca408},
}
db> select Log {action, timestamp, target_name, change};
{
  default::Log {
    action: 'insert',
    timestamp: <datetime>'2023-04-05T09:21:17.514089Z',
    target_name: {'Jonathan Harker', 'Mina Murray', 'Dracula'},
    change: {},
  },
  default::Log {
    action: 'update',
    timestamp: <datetime>'2023-04-05T09:35:30.389571Z',
    target_name: {'Jonathan', 'Mina'},
  }
}

```

(continues on next page)

(continued from previous page)

```
change: {'Jonathan Harker->Jonathan', 'Mina Murray->Mina'},  
},  
}
```

See also
<a href="#">SDL &gt; Triggers</a>
<a href="#">DDL &gt; Triggers</a>
<a href="#">Introspection &gt; Triggers</a>

## 2.14 Mutation rewrites

Mutation rewrites allow you to intercept database mutations (i.e., [inserts](#) and/or [updates](#)) and set the value of a property or link to the result of an expression you define. They can be defined in your schema.

Mutation rewrites are complementary to [triggers](#). While triggers are unable to modify the triggering object, mutation rewrites are built for that purpose.

Here's an example of a mutation rewrite that updates a property of a `Post` type to reflect the time of the most recent modification:

```
type Post {  
    required title: str;  
    required body: str;  
    modified: datetime {  
        rewrite insert, update using (datetime_of_statement())  
    }  
}
```

Every time a `Post` is updated, the mutation rewrite will be triggered, updating the `modified` property:

```
db> insert Post {  
...     title := 'One wierd trick to fix all your spelling errors'  
... };  
{default::Post {id: 19e024dc-d3b5-11ed-968c-37f5d0159e5f}}  
db> select Post {title, modified};  
{  
    default::Post {  
        title: 'One wierd trick to fix all your spelling errors',  
        modified: <datetime>'2023-04-05T13:23:49.488335Z',  
    },  
}  
db> update Post  
... filter .id = <uuid>'19e024dc-d3b5-11ed-968c-37f5d0159e5f'  
... set {title := 'One weird trick to fix all your spelling errors'};  
{default::Post {id: 19e024dc-d3b5-11ed-968c-37f5d0159e5f}}  
db> select Post {title, modified};  
{  
    default::Post {  
        title: 'One weird trick to fix all your spelling errors',  
        modified: <datetime>'2023-04-05T13:25:04.119641Z',  
    },
```

(continues on next page)

(continued from previous page)

```

},
}
```

In some cases, you will want different rewrites depending on the type of query. Here, we will add an `insert` rewrite and an `update` rewrite:

```

type Post {
    required title: str;
    required body: str;
    created: datetime {
        rewrite insert using (datetime_of_statement())
    }
    modified: datetime {
        rewrite update using (datetime_of_statement())
    }
}
```

With this schema, inserts will set the `Post` object's `created` property while updates will set the `modified` property:

```

db> insert Post {
...     title := 'One wierd trick to fix all your spelling errors'
... };
{default::Post {id: 19e024dc-d3b5-11ed-968c-37f5d0159e5f}}
db> select Post {title, created, modified};
{
    default::Post {
        title: 'One wierd trick to fix all your spelling errors',
        created: <datetime>'2023-04-05T13:23:49.488335Z',
        modified: {},
    },
}
db> update Post
... filter .id = <uuid>'19e024dc-d3b5-11ed-968c-37f5d0159e5f'
... set {title := 'One weird trick to fix all your spelling errors'};
{default::Post {id: 19e024dc-d3b5-11ed-968c-37f5d0159e5f}}
db> select Post {title, created, modified};
{
    default::Post {
        title: 'One weird trick to fix all your spelling errors',
        created: <datetime>'2023-04-05T13:23:49.488335Z',
        modified: <datetime>'2023-04-05T13:25:04.119641Z',
    },
}
```

---

**Note:** Each property may have a single `insert` and a single `update` mutation rewrite rule, or they may have a single rule that covers both.

---

## 2.14.1 Available variables

Inside the rewrite rule's expression, you have access to a few special values:

- `__subject__` refers to the object type with the new property and link values
- `__specified__` is a named tuple with a key for each property or link in the type and a boolean value indicating whether this value was explicitly set in the mutation
- `__old__` refers to the object type with the previous property and link values (available for update-only mutation rewrites)

Here are some examples of the special values in use. Maybe your blog hosts articles about particularly controversial topics. You could use `__subject__` to enforce a “cooling off” period before publishing a blog post:

```
type Post {
    required title: str;
    required body: str;
    publish_time: datetime {
        rewrite insert, update using (
            __subject__.publish_time ?? datetime_of_statement() +
            cal::to_relative_duration(days := 10)
        )
    }
}
```

Here we take the post's `publish_time` if set or the time the statement is executed and add 10 days to it. That should give our authors time to consider if they want to make any changes before a post goes live.

You can omit `__subject__` in many cases and achieve the same thing:

```
type Post {
    required title: str;
    required body: str;
    publish_time: datetime {
        rewrite insert, update using (
-            __subject__.publish_time ?? datetime_of_statement() +
+            .publish_time ?? datetime_of_statement() +
            cal::to_relative_duration(days := 10)
        )
    }
}
```

but only if the path prefix has not changed. In the following schema, for example, the `__subject__` in the rewrite rule is required, because in the context of the nested `select` query, the leading dot resolves from the `User` path:

```
type Post {
    required title: str;
    required body: str;
    author_email: str;
    author_name: str {
        rewrite insert, update using (
            (select User {name} filter .email = __subject__.author_email).name
        )
    }
}
```

(continues on next page)

(continued from previous page)

```
type User {
    name: str;
    email: str;
}
```

**Note:** Learn more about how this works in our documentation on [path resolution](#).

Using `__specified__`, we can determine which fields were specified in the mutation. This would allow us to track when a single property was last modified as in the `title_modified` property in this schema:

```
type Post {
    required title: str;
    required body: str;
    title_modified: datetime {
        rewrite update using (
            datetime_of_statement()
            if __specified__.title
            else __old__.title_modified
        )
    }
}
```

`__specified__.title` will be `true` if that value was set as part of the update, and this rewrite mutation rule will update `title_modified` to `datetime_of_statement()` in that case.

Another way you might use this is to set a default value but allow overriding:

```
type Post {
    required title: str;
    required body: str;
    modified: datetime {
        rewrite update using (
            datetime_of_statement()
            if not __specified__.modified
            else .modified
        )
    }
}
```

Here, we rewrite `modified` on updates to `datetime_of_statement()` unless `modified` was set in the update. In that case, we allow the specified value to be set. This is different from a `default` value because the rewrite happens on each update whereas a default value is applied only on insert of a new object.

Lastly, if we want to add an `author` property that can be set for each write and keep a history of all the authors, we can do this with the help of `__old__`:

```
type Post {
    required title: str;
    required body: str;
    author: str;
    all_authors: array<str> {
        default := <array<str>>[];
    }
}
```

(continues on next page)

(continued from previous page)

```

rewrite update using (
    __old__.all_authors
    ++ [__subject__.author]
);
}
}

```

On insert, our `all_authors` property will get initialized to an empty array of strings. We will rewrite updates to concatenate that array with an array containing the new author value.

## 2.14.2 Mutation rewrite as cached computed

Mutation rewrites can be used to effectively create a cached computed value as demonstrated with the `byline` property in this schema:

```

type Post {
    required title: str;
    required body: str;
    author: str;
    created: datetime {
        rewrite insert using (datetime_of_statement())
    }
    byline: str {
        rewrite insert, update using (
            'by ' ++
            __subject__.author ++
            ' on ' ++
            to_str(__subject__.created, 'Mon DD, YYYY')
        )
    }
}

```

The `byline` property will be updated on each insert or update, but the value will not need to be calculated at read time like a proper *computed property*.

See also
<a href="#">SDL &gt; Mutation rewrites</a>
<a href="#">DDL &gt; Mutation rewrites</a>
<a href="#">Introspection &gt; Mutation rewrites</a>

## 2.15 Inheritance

Inheritance is a crucial aspect of schema modeling in EdgeDB. Schema items can *extend* one or more parent types. When extending, the child (subclass) inherits the definition of its parents (superclass).

You can declare `abstract` object types, properties, links, constraints, and annotations.

- *Objects*
- *Properties*
- *Links*

- *Constraints*
- *Annotations*

## 2.15.1 Object types

Object types can *extend* other object types. The extending type (AKA the *subtype*) inherits all links, properties, indexes, constraints, etc. from its *supertypes*.

```
abstract type Animal {
    property species -> str;
}

type Dog extending Animal {
    property breed -> str;
}
```

```
abstract type Animal {
    species: str;
}

type Dog extending Animal {
    breed: str;
}
```

For details on querying polymorphic data, see [EdgeQL > Select > Polymorphic queries](#).

### 2.15.1.1 Multiple Inheritance

Object types can *extend more than one type* — that's called *multiple inheritance*. This mechanism allows building complex object types out of combinations of more basic types.

```
abstract type HasName {
    property first_name -> str;
    property last_name -> str;
}

abstract type HasEmail {
    property email -> str;
}

type Person extending HasName, HasEmail {
    property profession -> str;
}
```

```
abstract type HasName {
    first_name: str;
    last_name: str;
}

abstract type HasEmail {
    email: str;
```

(continues on next page)

(continued from previous page)

```

}

type Person extending HasName, HasEmail {
    profession: str;
}

```

### 2.15.1.2 Overloading

An object type can overload an inherited property or link. All overloaded declarations must be prefixed with the overloaded prefix to avoid unintentional overloads.

```

abstract type Person {
    property name -> str;
    multi link friends -> Person;
}

type Student extending Person {
    overloaded property name -> str {
        constraint exclusive;
    }
    overloaded multi link friends -> Student;
}

```

```

abstract type Person {
    name: str;
    multi friends: Person;
}

type Student extending Person {
    overloaded name: str {
        constraint exclusive;
    }
    overloaded multi friends: Student;
}

```

Overloaded fields cannot *generalize* the associated type; it can only make it *more specific* by setting the type to a subtype of the original or adding additional constraints.

### 2.15.2 Properties

Properties can be *concrete* (the default) or *abstract*. Abstract properties are declared independent of a source or target, can contain *annotations*, and can be marked as *readonly*.

```

abstract property title_prop {
    annotation title := 'A title.';
    readonly := false;
}

```

### 2.15.3 Links

It's possible to define abstract links that aren't tied to a particular *source* or *target*. Abstract links can be marked as readonly and contain annotations, property declarations, constraints, and indexes.

```
abstract link link_with_strength {
    property strength -> float64;
    index on (__subject__@strength);
}

type Person {
    multi link friends extending link_with_strength -> Person;
}
```

```
abstract link link_with_strength {
    strength: float64;
    index on (__subject__@strength);
}

type Person {
    multi friends: Person {
        extending link_with_strength;
    };
}
```

### 2.15.4 Constraints

Use `abstract` to declare reusable, user-defined constraint types.

```
abstract constraint in_range(min: anyreal, max: anyreal) {
    errmessage :=
        'Value must be in range [{min}, {max}].';
    using (max > __subject__ and __subject__ >= min);
}

type Player {
    property points -> int64 {
        constraint in_range(0, 100);
    }
}
```

```
abstract constraint in_range(min: anyreal, max: anyreal) {
    errmessage :=
        'Value must be in range [{min}, {max}].';
    using (max > __subject__ and __subject__ >= min);
}

type Player {
    points: int64 {
        constraint in_range(0, 100);
    }
}
```

## 2.15.5 Annotations

EdgeQL supports three annotation types by default: `title`, `description`, and `deprecated`. Use `abstract` annotation to declare custom user-defined annotation types.

```
abstract annotation admin_note;

type Status {
    annotation admin_note := 'system-critical';
    # more properties
}
```

By default, annotations defined on abstract types, properties, and links will not be inherited by their subtypes. To override this behavior, use the `inheritable` modifier.

```
abstract inheritable annotation admin_note;
```

## 2.16 Extensions

Extensions are the way EdgeDB adds more functionality. In principle, extensions could add new types, scalars, functions, etc., but, more importantly, they can add new ways of interacting with the database.

At the moment there are only two extensions available:

- `edgeql_http`: enables *EdgeQL over HTTP*
- `graphql`: enables *GraphQL*

See also
<a href="#">SDL &gt; Extensions</a>
<a href="#">DDL &gt; CREATE EXTENSION</a>
<a href="#">DDL &gt; DROP EXTENSION</a>

## 2.17 Future Behavior

Any time that we add new functionality to EdgeDB we strive to do it in the least disruptive way possible. Deprecation warnings, documentation and guides can help make these transitions smoother, but sometimes the changes are just too big, especially if they affect already existing functionality. It is often inconvenient dealing with these changes at the same time as upgrading to a new major version of EdgeDB. To help with this transition we introduce *future* specification.

The purpose of this specification is to provide a way to try out and ease into an upcoming feature before a major release. Sometimes enabling future behavior is necessary to fix some current issues. Other times enabling future behavior can simply provide a way to test out the feature before it gets released, to make sure that the current project codebase is compatible and well-behaved. It provides a longer timeframe for adopting a new feature and for catching bugs that arise from the change in behavior.

The *future* specification is intended to help with transitions between major releases. Once a feature is released this specification is no longer necessary to enable that feature and it will be removed from the schema during the upgrade process.

Once some behavior is available as a *future* all new *projects* enable this behavior by default when initializing an empty database. It is possible to explicitly disable the *future* feature by removing it from the schema, but it is not

recommended unless the feature is causing some issues which cannot be fixed otherwise. Existing projects don't change their behavior by default, the `future` specification needs to be added to the schema by the developer in order to gain early access to it.

At the moment there is only one `future` available:

- `nonrecursive_access_policies`: makes access policies *non-recursive* and simplifies policy interactions.

See also
<a href="#">SDL &gt; Future Behavior</a>
<a href="#">DDL &gt; CREATE FUTURE</a>
<a href="#">DDL &gt; DROP FUTURE</a>

## 2.18 vs SQL and ORMs

EdgeDB's approach to schema modeling builds upon the foundation of SQL while taking cues from modern tools like ORM libraries. Let's see how it stacks up.

### 2.18.1 Comparison to SQL

When using SQL databases, there's no convenient representation of the schema. Instead, the schema only exists as a series of `{CREATE|ALTER|DELETE} {TABLE| COLUMN}` commands, usually spread across several SQL migration scripts. There's no simple way to see the current state of your schema at a glance.

Moreover, SQL stores data in a *relational* way. Connections between tables are represented with foreign key constraints and `JOIN` operations are required to query across tables.

```
CREATE TABLE people (
    id          uuid PRIMARY KEY,
    name        text,
);
CREATE TABLE movies (
    id          uuid PRIMARY KEY,
    title       text,
    director_id uuid REFERENCES people(id)
);
```

In EdgeDB, connections between tables are represented with [Links](#).

```
type Movie {
    required property title -> str;
    required link director -> Person;
}

type Person {
    required property name -> str;
}
```

```
type Movie {
    required title: str;
    required director: Person;
}
```

(continues on next page)

(continued from previous page)

```
type Person {
    required name: str;
}
```

This approach makes it simple to write queries that traverse this link, no JOINs required.

```
select Movie {
    title,
    director: {
        name
    }
}
```

## 2.18.2 Comparison to ORMs

Object-relational mapping libraries are popular for a reason. They provide a way to model your schema and write queries in a way that feels natural in the context of modern, object-oriented programming languages. But ORMs have downsides too.

- **Lock-in.** Your schema is strongly coupled to the ORM library you are using. More generally, this also locks you into using a particular programming language.
- Most ORMs have more **limited querying capabilities** than the query languages they abstract.
- Many ORMs produce **suboptimal queries** that can have serious performance implications.
- **Migrations** can be difficult. Since most ORMs aim to be the single source of truth for your schema, they necessarily must provide some sort of migration tool. These migration tools are maintained by the contributors to the ORM library, not the maintainers of the database itself. Quality control and long-term maintenance is not always guaranteed.

From the beginning, EdgeDB was designed to incorporate the best aspects of ORMs — declarative modeling, object-oriented APIs, and intuitive querying — without the drawbacks.

## 2.19 Introspection

All of the schema information in EdgeDB is stored in the schema `module` and is accessible via *introspection queries*.

All the introspection types are themselves extending `BaseObjectType`, so they are also subject to introspection *as object types*. The following query will give a list of all the types used in introspection:

```
select name := schema::ObjectType.name
filter name like 'schema::%';
```

There's also a couple of ways of getting the introspection type of a particular expression. Any `Object` has a `__type__` link to the `schema::ObjectType`. For scalars there's the `introspect` and `typeof` operators that can be used to get the type of an expression.

Finally, the command `describe` can be used to get information about EdgeDB types in a variety of human-readable formats.

## 2.19.1 Object types

This section describes introspection of *object types*.

Introspection of the schema::ObjectType:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::ObjectType';
{
    Object {
        name: 'schema::ObjectType',
        links: {
            Object { name: '__type__' },
            Object { name: 'annotations' },
            Object { name: 'bases' },
            Object { name: 'constraints' },
            Object { name: 'indexes' },
            Object { name: 'links' },
            Object { name: 'ancestors' },
            Object { name: 'pointers' },
            Object { name: 'properties' }
        },
        properties: {
            Object { name: 'id' },
            Object { name: 'abstract' },
            Object { name: 'name' }
        }
    }
}
```

Consider the following schema:

```
type Addressable {
    property address -> str;
}

type User extending Addressable {
    # define some properties and a link
    required property name -> str;

    multi link friends -> User;

    # define an index for User based on name
    index on (.name);
}
```

Introspection of User:

```
db> with module schema
... select ObjectType {
...     name,
...     abstract,
...     bases: { name },
...     ancestors: { name },
...     annotations: { name, @value },
...     links: {
...         name,
...         cardinality,
...         required,
...         target: { name },
...     },
...     properties: {
...         name,
...         cardinality,
...         required,
...         target: { name },
...     },
...     constraints: { name },
...     indexes: { expr },
... }
... filter .name = 'default::User';
{
    Object {
        name: 'default::User',
        abstract: false,
        bases: {Object { name: 'default::Addressable' }},
        ancestors: {
            Object { name: 'std::BaseObject' },
            Object { name: 'std::Object' },
            Object { name: 'default::Addressable' }
        },
        annotations: {},
        links: {
            Object {
                name: '__type__',
                cardinality: 'One',
                required: {},
                target: Object { name: 'schema::Type' }
            },
            Object {
                name: 'friends',
                cardinality: 'Many',
                required: false,
                target: Object { name: 'default::User' }
            }
        },
        properties: {
            Object {
                name: 'address',
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
        cardinality: 'One',
        required: false,
        target: Object { name: 'std::str' }
    },
    Object {
        name: 'id',
        cardinality: 'One',
        required: true,
        target: Object { name: 'std::uuid' }
    },
    Object {
        name: 'name',
        cardinality: 'One',
        required: true,
        target: Object { name: 'std::str' }
    }
},
constraints: {},
indexes: {
    Object {
        expr: '.name'
    }
}
}
```

**See also**

- [Schema > Object types](#)
- [SDL > Object types](#)
- [DDL > Object types](#)
- [Cheatsheets > Object types](#)

## 2.19.2 Scalar types

This section describes introspection of *scalar types*.

Introspection of the schema::ScalarType:

```
db> with module schema
... select ObjectType {
...     name,
...     links: [
...         {
...             name,
...             properties: [
...                 {
...                     name,
...                 }
...             ]
...         }
...     ]
...     filter .name = 'schema::ScalarType';
{
```

(continues on next page)

(continued from previous page)

```

name: 'schema::ScalarType',
links: {
    Object { name: '__type__' },
    Object { name: 'annotations' },
    Object { name: 'bases' },
    Object { name: 'constraints' },
    Object { name: 'ancestors' }
},
properties: {
    Object { name: 'default' },
    Object { name: 'enum_values' },
    Object { name: 'id' },
    Object { name: 'abstract' },
    Object { name: 'name' }
}
}
}
}

```

Introspection of the built-in scalar `str`:

```

db> with module schema
... select ScalarType {
...     name,
...     default,
...     enum_values,
...     abstract,
...     bases: { name },
...     ancestors: { name },
...     annotations: { name, @value },
...     constraints: { name },
... }
... filter .name = 'std::str';
{
    Object {
        name: 'std::str',
        default: {},
        enum_values: {},
        abstract: {},
        bases: {Object { name: 'std::anyscalar' }},
        ancestors: {Object { name: 'std::anyscalar' }},
        annotations: {},
        constraints: {}
    }
}

```

For an *enumerated scalar type*, consider the following:

```
scalar type Color extending enum<Red, Green, Blue>;
```

Introspection of the enum scalar `Color`:

```

db> with module schema
... select ScalarType {

```

(continues on next page)

(continued from previous page)

```

...
    name,
    default,
    enum_values,
    abstract,
    bases: { name },
    ancestors: { name },
    annotations: { name, @value },
    constraints: { name },
}
filter .name = 'default::Color';
{
    Object {
        name: 'default::Color',
        default: {},
        enum_values: ['Red', 'Green', 'Blue'],
        abstract: {},
        bases: {Object { name: 'std::anyenum' }},
        ancestors: {
            Object { name: 'std::anyscalar' },
            Object { name: 'std::anyenum' }
        },
        annotations: {},
        constraints: {}
    }
}

```

### 2.19.3 Collection types

This section describes introspection of *collection types*.

#### 2.19.3.1 Array

Introspection of the `schema::Array`:

```

db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Array';
{
    Object {
        name: 'schema::Array',
        links: {
            Object { name: '__type__' },

```

(continues on next page)

(continued from previous page)

```

    Object { name: 'element_type' }
},
properties: {
    Object { name: 'id' },
    Object { name: 'name' }
}
}
}
```

For a type with an `array` property, consider the following:

```

type User {
    required property name -> str;
    property favorites -> array<str>;
}
```

Introspection of the User with emphasis on properties:

```

db> with module schema
... select ObjectType {
...     name,
...     properties: {
...         name,
...         target: {
...             name,
...             [is Array].element_type: { name },
...         },
...     },
... }
... filter .name = 'default::User';
{
    Object {
        name: 'default::User',
        properties: {
            Object {
                name: 'favorites',
                target: Object {
                    name: 'array',
                    element_type: Object { name: 'std::str' }
                }
            },
            ...
        }
    }
}
```

### 2.19.3.2 Tuple

Introspection of the schema::Tuple:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Tuple';
{
    Object {
        name: 'schema::Tuple',
        links: {
            Object { name: '__type__' },
            Object { name: 'element_types' }
        },
        properties: {
            Object { name: 'id' },
            Object { name: 'name' }
        }
    }
}
```

For example, below is an introspection of the return type of the `sys::get_version()` function:

```
db> with module schema
... select `Function` {
...     return_type[is Tuple]: {
...         element_types: {
...             name,
...             type: { name }
...         } order by .num
...     }
... }
... filter .name = 'sys::get_version';
{
    Object {
        return_type: Object {
            element_types: {
                Object {
                    name: 'major',
                    type: Object {
                        name: 'std::int64'
                    }
                },
                Object {
                    name: 'minor',
                    type: Object {

```

(continues on next page)

(continued from previous page)

```
        name: 'std::int64'
    }
},
Object {
    name: 'stage',
    type: Object {
        name: 'sys::VersionStage'
    }
},
Object {
    name: 'stage_no',
    type: Object {
        name: 'std::int64'
    }
},
Object {
    name: 'local',
    type: Object { name: 'array' }
}
}
}
```

## 2.19.4 Functions

This section describes introspection of *functions*.

#### Introspection of the schema::Function:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...         },
...     properties: {
...         name,
...         }
... }
... filter .name = 'schema::Function';
{
    Object {
        name: 'schema::Function',
        links: {
            Object { name: '__type__' },
            Object { name: 'annotations' },
            Object { name: 'params' },
            Object { name: 'return_type' }
        },
        properties: {
```

(continues on next page)

(continued from previous page)

```

Object { name: 'id' },
Object { name: 'name' },
Object { name: 'return_typemod' }
}
}
}
```

Since params are quite important to functions, here's their structure:

```

db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Parameter';
{
    Object {
        name: 'schema::Parameter',
        links: {
            Object { name: '__type__' },
            Object { name: 'type' }
        },
        properties: {
            Object { name: 'default' },
            Object { name: 'id' },
            Object { name: 'kind' },
            Object { name: 'name' },
            Object { name: 'num' },
            Object { name: 'typemod' }
        }
    }
}
```

Introspection of the built-in `count()`:

```

db> with module schema
... select `Function` {
...     name,
...     annotations: { name, @value },
...     params: {
...         kind,
...         name,
...         num,
...         typemod,
...         type: { name },
...         default,
...     },
... }
```

(continues on next page)

(continued from previous page)

```

...
    return_typemod,
...
    return_type: { name },
...
}
... filter .name = 'std::count';
{
    Object {
        name: 'std::count',
        annotations: {},
        params: {
            Object {
                kind: 'PositionalParam',
                name: 's',
                num: 0,
                typemod: 'SetOfType',
                type: Object { name: 'anytype' },
                default: {}
            }
        },
        return_typemod: 'SingletonType',
        return_type: Object { name: 'std::int64' }
    }
}
}

```

See also
<i>Schema &gt; Functions</i>
<i>SDL &gt; Functions</i>
<i>DDL &gt; Functions</i>
<i>Reference &gt; Function calls</i>
<i>Cheatsheets &gt; Functions</i>

## 2.19.5 Triggers

This section describes introspection of *triggers*.

Introspection of `schema::Trigger`:

```

db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... } filter .name = 'schema::Trigger';
{
    schema::ObjectType {
        name: 'schema::Trigger',
        links: {
            schema::Link {name: 'subject'},

```

(continues on next page)

(continued from previous page)

```

schema::Link {name: '__type__'},
schema::Link {name: 'ancestors'},
schema::Link {name: 'bases'},
schema::Link {name: 'annotations'}
},
properties: {
    schema::Property {name: 'inherited_fields'},
    schema::Property {name: 'computed_fields'},
    schema::Property {name: 'builtin'},
    schema::Property {name: 'internal'},
    schema::Property {name: 'name'},
    schema::Property {name: 'id'},
    schema::Property {name: 'abstract'},
    schema::Property {name: 'is_abstract'},
    schema::Property {name: 'final'},
    schema::Property {name: 'is_final'},
    schema::Property {name: 'timing'},
    schema::Property {name: 'kinds'},
    schema::Property {name: 'scope'},
    schema::Property {name: 'expr'},
},
},
}

```

Introspection of a trigger named `log_insert` on the `User` type:

```

db> with module schema
... select Trigger {
...     name,
...     kinds,
...     timing,
...     scope,
...     expr,
...     subject: {
...         name
...     }
... } filter .name = 'log_insert';
{
    schema::Trigger {
        name: 'log_insert',
        kinds: {Insert},
        timing: After,
        scope: Each,
        expr: 'insert default::Log { action := \\'insert\\', target_name := __new__.name }',
        subject: schema::ObjectType {name: 'default::User'},
    },
}

```

See also
<a href="#">Schema &gt; Triggers</a>
<a href="#">SDL &gt; Triggers</a>
<a href="#">DDL &gt; Triggers</a>

## 2.19.6 Mutation rewrites

This section describes introspection of *mutation rewrites*.

Introspection of the `schema::Rewrite`:

```
db> select schema::ObjectType {
...     name,
...     links: {
...         name
...     },
...     properties: {
...         name
...     }
... } filter .name = 'schema::Rewrite';
{
    schema::ObjectType {
        name: 'schema::Rewrite',
        links: {
            schema::Link {name: 'subject'},
            schema::Link {name: '__type__'},
            schema::Link {name: 'ancestors'},
            schema::Link {name: 'bases'},
            schema::Link {name: 'annotations'}
        },
        properties: {
            schema::Property {name: 'inherited_fields'},
            schema::Property {name: 'computed_fields'},
            schema::Property {name: 'builtin'},
            schema::Property {name: 'internal'},
            schema::Property {name: 'name'},
            schema::Property {name: 'id'},
            schema::Property {name: 'abstract'},
            schema::Property {name: 'is_abstract'},
            schema::Property {name: 'final'},
            schema::Property {name: 'is_final'},
            schema::Property {name: 'kind'},
            schema::Property {name: 'expr'}
        },
    },
},
}
```

Introspection of all properties in the default schema with a mutation rewrite:

```
db> select schema::ObjectType {
...     name,
...     properties := (
...         select .properties {
...             name,
...             rewrites: {
...                 kind
...             }
...         } filter exists .rewrites
...     )
... }
```

(continues on next page)

(continued from previous page)

```

... } filter .name ilike 'default::%'
... and exists .properties.rewrites;
{
    schema::ObjectType {
        name: 'default::Post',
        properties: {
            schema::Property {
                name: 'created',
                rewrites: {
                    schema::Rewrite {
                        kind: Insert
                    }
                }
            },
            schema::Property {
                name: 'modified',
                rewrites: {
                    schema::Rewrite {
                        kind: Insert
                    },
                    schema::Rewrite {
                        kind: Update
                    }
                }
            },
            ...
        },
    }
}

```

Introspection of all rewrites, including the type of query (kind), rewrite expression, and the object and property they are on:

```

db> select schema::Rewrite {
...     subject := (
...         select .subject {
...             name,
...             source: {
...                 name
...             }
...         }
...     ),
...     kind,
...     expr
... );
{
    schema::Rewrite {
        subject: schema::Property {
            name: 'created',
            source: schema::ObjectType {
                name: 'default::Post'
            }
        },
    }
}

```

(continues on next page)

(continued from previous page)

```

kind: Insert,
expr: 'std::datetime_of_statement()'
},
schema::Rewrite {
subject: schema::Property {
name: 'modified',
source: schema::ObjectType {
name: 'default::Post'
}
},
kind: Insert,
expr: 'std::datetime_of_statement()'
},
schema::Rewrite {
subject: schema::Property {
name: 'modified',
source: schema::ObjectType {
name: 'default::Post'
}
},
kind: Update,
expr: 'std::datetime_of_statement()'
},
}
}

```

Introspection of all rewrites on a `default::Post` property named `modified`:

```

db> select schema::Rewrite {kind, expr}
... filter .subject.source.name = 'default::Post'
... and .subject.name = 'modified';
{
  schema::Rewrite {
    kind: Insert,
    expr: 'std::datetime_of_statement()'
  },
  schema::Rewrite {
    kind: Update,
    expr: 'std::datetime_of_statement()'
  }
}

```

See also
<a href="#">Schema &gt; Mutation rewrites</a>
<a href="#">SDL &gt; Mutation rewrites</a>
<a href="#">DDL &gt; Mutation rewrites</a>

## 2.19.7 Indexes

This section describes introspection of *indexes*.

Introspection of the `schema::Index`:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Index';
{
    Object {
        name: 'schema::Index',
        links: {Object { name: '__type__' }},
        properties: {
            Object { name: 'expr' },
            Object { name: 'id' },
            Object { name: 'name' }
        }
    }
}
```

Consider the following schema:

```
type Addressable {
    property address -> str;
}

type User extending Addressable {
    # define some properties and a link
    required property name -> str;

    multi link friends -> User;

    # define an index for User based on name
    index on (.name);
}
```

Introspection of `User.name` index:

```
db> with module schema
... select Index {
...     expr,
... }
... filter .expr like '%.name';
{
    Object {
```

(continues on next page)

(continued from previous page)

```

        expr: '.name'
    }
}

```

For introspection of the index within the context of its host type see [object type introspection](#).

See also
<a href="#">Schema &gt; Indexes</a>
<a href="#">SDL &gt; Indexes</a>
<a href="#">DDL &gt; Indexes</a>

## 2.19.8 Constraints

This section describes introspection of [constraints](#).

Introspection of the `schema::Constraint`:

```

db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Constraint';
{
    Object {
        name: 'schema::Constraint',
        links: {
            Object { name: '__type__' },
            Object { name: 'args' },
            Object { name: 'annotations' },
            Object { name: 'bases' },
            Object { name: 'ancestors' },
            Object { name: 'params' },
            Object { name: 'return_type' },
            Object { name: 'subject' }
        },
        properties: {
            Object { name: 'errmessage' },
            Object { name: 'expr' },
            Object { name: 'finalexpr' },
            Object { name: 'id' },
            Object { name: 'abstract' },
            Object { name: 'name' },
            Object { name: 'return_typemod' },
            Object { name: 'subjectexpr' }
        }
}

```

(continues on next page)

(continued from previous page)

```

    }
}
```

Consider the following schema:

```
scalar type maxex_100 extending int64 {
    constraint max_ex_value(100);
}
```

Introspection of the scalar `maxex_100` with focus on the constraint:

```
db> with module schema
... select ScalarType {
...     name,
...     constraints: {
...         name,
...         expr,
...         annotations: { name, @value },
...         subject: { name },
...         params: { name, @value, type: { name } },
...         return_typemod,
...         return_type: { name },
...         errmessage,
...     },
... }
... filter .name = 'default::maxex_100';
{
    Object {
        name: 'default::maxex_100',
        constraints: {
            Object {
                name: 'std::max_ex_value',
                expr: '(__subject__ <= max)',
                annotations: {},
                subject: Object { name: 'default::maxex_100' },
                params: {
                    Object {
                        name: 'max',
                        type: Object { name: 'anytype' },
                        @value: '100'
                    }
                },
                return_typemod: 'SingletonType',
                return_type: Object { name: 'std::bool' }
                errmessage: '{__subject__} must be less ...',
            }
        }
    }
}
```

See also
<a href="#">Schema &gt; Constraints</a>
<a href="#">SDL &gt; Constraints</a>
<a href="#">DDL &gt; Constraints</a>
<a href="#">Standard Library &gt; Constraints</a>

## 2.19.9 Operators

This section describes introspection of EdgeDB operators. Much like functions, operators have parameters and return types as well as a few other features.

Introspection of the `schema::Operator`:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Operator';
{
    Object {
        name: 'schema::Operator',
        links: {
            Object { name: '__type__' },
            Object { name: 'annotations' },
            Object { name: 'params' },
            Object { name: 'return_type' }
        },
        properties: {
            Object { name: 'id' },
            Object { name: 'name' },
            Object { name: 'operator_kind' },
            Object { name: 'return_typemod' }
        }
    }
}
```

Since `params` are quite important to operators, here's their structure:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
```

(continues on next page)

(continued from previous page)

```

...
}
...
filter .name = 'schema::Parameter';
{
  Object {
    name: 'schema::Parameter',
    links: {
      Object { name: '__type__' },
      Object { name: 'type' }
    },
    properties: {
      Object { name: 'default' },
      Object { name: 'id' },
      Object { name: 'kind' },
      Object { name: 'name' },
      Object { name: 'num' },
      Object { name: 'typemod' }
    }
  }
}

```

Introspection of the `and` operator:

```

db> with module schema
... select Operator {
...   name,
...   operator_kind,
...   annotations: { name, @value },
...   params: {
...     kind,
...     name,
...     num,
...     typemod,
...     type: { name },
...     default,
...   },
...   return_typemod,
...   return_type: { name },
... }
... filter .name = 'std::AND';
{
  Object {
    name: 'std::AND',
    operator_kind: 'Infix',
    annotations: {},
    params: {
      Object {
        kind: 'PositionalParam',
        name: 'a',
        num: 0,
        typemod: 'SingletonType',
        type: Object { name: 'std::bool' },
      }
    }
  }
}

```

(continues on next page)

(continued from previous page)

```

        default: {}
    },
Object {
    kind: 'PositionalParam',
    name: 'b',
    num: 1,
    typemod: 'SingletonType',
    type: Object { name: 'std::bool' },
    default: {}
}
},
return_typemod: 'SingletonType',
return_type: Object { name: 'std::bool' }
}
}
}
```

## 2.19.10 Casts

This section describes introspection of EdgeDB *type casts*. Features like whether the casts are implicit can be discovered by introspecting `schema::Cast`.

Introspection of the `schema::Cast`:

```
db> with module schema
... select ObjectType {
...     name,
...     links: {
...         name,
...     },
...     properties: {
...         name,
...     }
... }
... filter .name = 'schema::Cast';
{
Object {
    name: 'schema::Cast',
    links: {
        Object { name: '__type__' },
        Object { name: 'from_type' },
        Object { name: 'to_type' }
    },
    properties: {
        Object { name: 'allow_assignment' },
        Object { name: 'allow_implicit' },
        Object { name: 'id' },
        Object { name: 'name' }
    }
}
```

Introspection of the possible casts from `std::int64` to other types:

```
db> with module schema
... select Cast {
...     allow_assignment,
...     allow_implicit,
...     to_type: { name },
... }
... filter .from_type.name = 'std::int64'
... order by .to_type.name;
{
    Object {
        allow_assignment: false,
        allow_implicit: true,
        to_type: Object { name: 'std::bigint' }
    },
    Object {
        allow_assignment: false,
        allow_implicit: true,
        to_type: Object { name: 'std::decimal' }
    },
    Object {
        allow_assignment: true,
        allow_implicit: false,
        to_type: Object { name: 'std::float32' }
    },
    Object {
        allow_assignment: false,
        allow_implicit: true,
        to_type: Object { name: 'std::float64' }
    },
    Object {
        allow_assignment: true,
        allow_implicit: false,
        to_type: Object { name: 'std::int16' }
    },
    Object {
        allow_assignment: true,
        allow_implicit: false,
        to_type: Object { name: 'std::int32' }
    },
    Object {
        allow_assignment: false,
        allow_implicit: false,
        to_type: Object { name: 'std::json' }
    },
    Object {
        allow_assignment: false,
        allow_implicit: false,
        to_type: Object { name: 'std::str' }
    }
}
```

The `allow_implicit` property tells whether this is an *implicit cast* in all contexts (such as when determining the type of a set of mixed literals or resolving the argument types of functions or operators if there's no exact match). For

example, a literal 1 is an `int64` and it is implicitly cast into a `bigint` or `float64` if it is added to a set containing either one of those types:

```
db> select {1, 2n};
{1n, 2n}
db> select {1, 2.0};
{1.0, 2.0}
```

What happens if there's no implicit cast between a couple of scalars in this type of example? EdgeDB checks whether there's a scalar type such that all of the set elements can be implicitly cast into that:

```
db> select introspect (typeof {<int64>1, <float32>2}).name;
{'std::float64'}
```

The scalar types `int64` and `float32` cannot be implicitly cast into each other, but they both can be implicitly cast into `float64`.

The `allow_assignment` property tells whether this is an implicit cast during assignment if a more general *implicit cast* is not allowed. For example, consider the following type:

```
type Example {
    property p_int16 -> int16;
    property p_float32 -> float32;
    property p_json -> json;
}
```

```
db> insert Example {
...     p_int16 := 1,
...     p_float32 := 2
... };
{Object { id: <uuid>'...' }}
db> insert Example {
...     p_json := 3 # assignment cast to json not allowed
... };
InvalidPropertyTargetException: invalid target for property
'p_json' of object type 'default::Example': 'std::int64'
(expecting 'std::json')
```

EdgeDB schemas are declared using **SDL** (EdgeDB's Schema Definition Language).

## 2.20 SDL

Your schema is defined inside `.esdl` files. It's common to define your entire schema in a single file called `default.esdl`, but you can split it across multiple files if you wish.

By convention, your schema files should live in a directory called `dbschema` in the root of your project.

```
# dbschema/default.esdl

type Movie {
    required property title -> str;
    required link director -> Person;
}
```

(continues on next page)

(continued from previous page)

```
type Person {
    required property name -> str;
}
```

```
# dbschema/default.esdl

type Movie {
    required title: str;
    required director: Person;
}

type Person {
    required name: str;
}
```

---

**Important:** Syntax highlighter packages/extensions for .esdl files are available for Visual Studio Code, Sublime Text, Atom, and Vim.

---

## 2.21 Migrations

EdgeDB's baked-in migration system lets you painlessly evolve your schema over time. Just update the contents of your .esdl file(s) and use the EdgeDB CLI to *create* and *apply* migrations.

```
$ edgedb migration create
Created dbschema/migrations/00001.esdl
$ edgedb migrate
Applied dbschema/migrations/00001.esdl.
```

For a full guide on migrations, refer to the [Creating and applying migrations](#) guide.

---

**Important:** A migration consists of a sequence of *imperative* schema-modifying commands like `create type`, `alter property`, etc. Collectively these commands are known as *DDL* (*data definition language*). We recommend using SDL and the migration system when building applications, however you're free to use DDL directly if you prefer.

---

## 2.22 Terminology

### Instance

An EdgeDB **instance** is a collection of databases that store their data in a shared directory, listen for queries on a particular port, and are managed by a running EdgeDB process. Instances can be created, started, stopped, and destroyed locally with the [EdgeDB CLI](#).

## Database

Each instance can contain several **databases**, each with a unique name. At the time of creation, all instances contain a single default database called `edgedb`. All incoming queries are executed against it unless otherwise specified.

## Module

Each database has a schema consisting of several **modules**, each with a unique name. Modules can be used to organize large schemas into logical units. In practice, though, most users put their entire schema inside a single module called `default`.

```
module default {
    # declare types here
}
```

You may define nested modules using the following syntax:

```
module dracula {
    type Person {
        required property name -> str;
        multi link places_visited -> City;
        property strength -> int16;
    }

    module combat {
        function fight(one: Person, two: Person) -> str
        using (
            (one.name ?? 'Fighter 1') ++ ' wins!'
            IF (one.strength ?? 0) > (two.strength ?? 0)
            ELSE (two.name ?? 'Fighter 2') ++ ' wins!'
        );
    }
}
```

Here we have a `dracula` module containing a `Person` type. Nested in the `dracula` module we have a `combat` module which will be used for all the combat functionality for our game based on Bram Stoker's Dracula we built in the [Easy EdgeDB textbook](#).

---

### Note: Name resolution

When referencing schema objects from another module, you must use a *fully-qualified* name in the form `module_name::object_name`.

---

The following module names are reserved by EdgeDB and contain pre-defined types, utility functions, and operators.

- `std`: standard types, functions, and operators in the *standard library*
- `math`: algebraic and statistical *functions*
- `cal`: local (non-timezone-aware) and relative date/time *types and functions*
- `schema`: types describing the *introspection* schema
- `sys`: system-wide entities, such as user roles and *databases*
- `cfg`: configuration and settings

You can chain together module names in a fully-qualified name to traverse a tree of nested modules. For example, to call the `fight` function in the nested module example above, you would use `dracula::combat::fight(<arguments>)`.

## EDGSQL

### 3.1 Literals

EdgeQL is *inextricably tied* to EdgeDB's rigorous type system. Below is an overview of how to declare a literal value of each *primitive type*. Click a link in the left column to jump to the associated section.

<i>String</i>	<code>str</code>
<i>Boolean</i>	<code>bool</code>
<i>Numbers</i>	<code>int16 int32 int64 float32 float64 bigint decimal</code>
<i>UUID</i>	<code>uuid</code>
<i>Enums</i>	<code>enum&lt;X, Y, Z&gt;</code>
<i>Dates and times</i>	<code>datetime duration cal::local_datetime cal::local_date cal::local_time cal::relative_duration</code>
<i>Durations</i>	<code>duration cal::relative_duration cal::date_duration</code>
<i>Bytes</i>	<code>bytes</code>
<i>Arrays</i>	<code>array&lt;x&gt;</code>
<i>Tuples</i>	<code>tuple&lt;x, y, ...&gt; or tuple&lt;foo: x, bar: y, ...&gt;</code>
<i>JSON</i>	<code>json</code>

#### 3.1.1 Strings

The `str` type is a variable-length string of Unicode characters. A string can be declared with either single or double quotes.

```
db> select 'i edgedb';
{i edgedb}
db> select "hello there!";
{'hello there!'}
db> select 'hello\nthere!';
{'hello
there!'}
db> select 'hello
... there!';
{'hello
there!'}
db> select r'hello
... there!'; # multiline
{'hello
there!'}

---


```

There is a special syntax for declaring “raw strings”. Raw strings treat the backslash \ as a literal character instead of an escape character.

```
db> select r'hello\nthere'; # raw string
{r'hello\\nthere'}
db> select $$one
... two
... three$$; # multiline raw string
{'one
two
three'}
db> select $label$You can add an interstitial label
... if you need to use "$$" in your string.$label$;
{
    'You can add an interstitial label
    if you need to use " $$ " in your string.',
}
```

EdgeQL contains a set of built-in functions and operators for searching, comparing, and manipulating strings.

```
db> select 'hellothere'[5:10];
{'there'}
db> select 'hello' ++ 'there';
{'hellothere'}
db> select len('hellothere');
{10}
db> select str_trim(' hello there ');
{'hello there'}
db> select str_split('hello there', ' ');
{['hello', 'there']}
```

For a complete reference on strings, see [Standard Library > String](#) or click an item below.

Indexing and slicing	<code>str[i]</code> <code>str[from:to]</code>
Concatenation	<code>str ++ str</code>
Utilities	<code>len()</code>
Transformation functions	<code>str_split()</code> <code>str_lower()</code> <code>str_upper()</code> <code>str_title()</code> <code>str_pad_start()</code> <code>str_pad_end()</code> <code>str_trim()</code> <code>str_trim_start()</code> <code>str_trim_end()</code> <code>str_repeat()</code>
Comparison operators	<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>
Search	<code>contains()</code> <code>find()</code>
Pattern matching and regexes	<code>str like pattern</code> <code>str ilike pattern</code> <code>re_match()</code> <code>re_match_all()</code> <code>re_replace()</code> <code>re_test()</code>

### 3.1.2 Booleans

The `bool` type represents a true/false value.

```
db> select true;
{true}
db> select false;
{false}
```

EdgeDB provides a set of operators that operate on boolean values.

Comparison operators	<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>
Logical operators	<code>or and not</code>
Aggregation	<code>all() any()</code>

### 3.1.3 Numbers

There are several numerical types in EdgeDB's type system.

<code>int16</code>	16-bit integer
<code>int32</code>	32-bit integer
<code>int64</code>	64-bit integer
<code>float32</code>	32-bit floating point number
<code>float64</code>	64-bit floating point number
<code>bigint</code>	Arbitrary precision integer.
<code>decimal</code>	Arbitrary precision number.

Number literals that *do not* contain a decimal are interpreted as `int64`. Numbers containing decimals are interpreted as `float64`. The `n` suffix designates a number with *arbitrary precision*: either `bigint` or `decimal`.

Syntax	Inferred type
<code>select 3;</code>	<code>int64</code>
<code>select 3.14;</code>	<code>float64</code>
<code>select 314e-2;</code>	<code>float64</code>
<code>select 42n;</code>	<code>bigint</code>
<code>select 42.0n;</code>	<code>decimal</code>
<code>select 42e+100n;</code>	<code>decimal</code>

To declare an `int16`, `int32`, or `float32`, you must provide an explicit type cast. For details on type casting, see [Casting](#).

Syntax	Type
<code>select &lt;int16&gt;1234;</code>	<code>int16</code>
<code>select &lt;int32&gt;123456;</code>	<code>int32</code>
<code>select &lt;float32&gt;123.456;</code>	<code>float32</code>

EdgeQL includes a full set of arithmetic and comparison operators. Parentheses can be used to indicate the order-of-operations or visually group subexpressions; this is true across all EdgeQL queries.

```
db> select 5 > 2;
{true}
db> select 2 + 2;
{4}
db> select 2 ^ 10;
{1024}
db> select (1 + 1) * 2 / (3 + 8);
{0.36363636363636365}
```

EdgeQL provides a comprehensive set of built-in functions and operators on numerical data.

Comparison operators	= != ?= ?!= < > <= >=
Arithmetic	+ - * / // % ^
Statistics	sum() min() max() math::mean() math::stddev() math::stddev_pop() math::var() math::var_pop()
Math	round() math::abs() math::ceil() math::floor() math::ln() math::lg() math::log()
Random number	random()

### 3.1.4 UUID

The `uuid` type is commonly used to represent object identifiers. UUID literal must be explicitly cast from a string value matching the UUID specification.

```
db> select <uuid>'a5ea6360-75bd-4c20-b69c-8f317b0d2857';
{a5ea6360-75bd-4c20-b69c-8f317b0d2857}
```

Generate a random UUID.

```
db> select uuid_generate_v1mc();
{b4d94e6c-3845-11ec-b0f4-93e867a589e7}
```

### 3.1.5 Enums

Enum types must be *declared in your schema*.

```
scalar type Color extending enum<Red, Green, Blue>;
```

Once declared, an enum literal can be declared with dot notation, or by casting an appropriate string literal:

```
db> select Color.Red;
{Red}
db> select <Color>"Red";
{Red}
```

### 3.1.6 Dates and times

EdgeDB's typesystem contains several temporal types.

<code>datetime</code>	Timezone-aware point in time
<code>cal::local_datetime</code>	Date and time w/o timezone
<code>cal::local_date</code>	Date type
<code>cal::local_time</code>	Time type

All temporal literals are declared by casting an appropriately formatted string.

```
db> select <datetime>'1999-03-31T15:17:00Z';
{<datetime>'1999-03-31T15:17:00Z'}
db> select <datetime>'1999-03-31T17:17:00+02';
{<datetime>'1999-03-31T15:17:00Z'}
db> select <cal::local_datetime>'1999-03-31T15:17:00';
{<cal::local_datetime>'1999-03-31T15:17:00'}
db> select <cal::local_date>'1999-03-31';
{<cal::local_date>'1999-03-31'}
db> select <cal::local_time>'15:17:00';
{<cal::local_time>'15:17:00'}
```

EdgeQL supports a set of functions and operators on datetime types.

Comparison operators	<code>= != ?= ?!= &lt;&gt; &lt;= &gt;=</code>
Arithmetic	<code>dt + dt</code> <code>dt - dt</code>
String parsing	<code>to_datetime()</code> <code>cal::to_local_datetime()</code> <code>cal::to_local_date()</code> <code>cal::to_local_time()</code>
Component extraction	<code>datetime_get()</code> <code>cal::time_get()</code> <code>cal::date_get()</code>
Truncation	<code>datetime_truncate()</code>
System timestamps	<code>datetime_current()</code> <code>datetime_of_transaction()</code> <code>datetime_of_statement()</code>

### 3.1.7 Durations

EdgeDB's type system contains three duration types.

<code>duration</code>	Exact duration
<code>cal::relative_duration</code>	Duration in relative units
<code>cal::date_duration</code>	Duration in months and days only

### 3.1.7.1 Exact durations

The `duration` type represents *exact* durations that can be represented by some fixed number of microseconds. It can be negative and it supports units of `microseconds`, `milliseconds`, `seconds`, `minutes`, and `hours`.

```
db> select <duration>'45.6 seconds';
{<duration>'0:00:45.6'}
db> select <duration>'-15 microseconds';
{<duration>'-0:00:00.000015'}
db> select <duration>'5 hours 4 minutes 3 seconds';
{<duration>'5:04:03'}
db> select <duration>'8760 hours'; # about a year
{<duration>'8760:00:00'}
```

All temporal units beyond hour no longer correspond to a fixed duration of time; the length of a day/month/year/etc changes based on daylight savings time, the month in question, leap years, etc.

### 3.1.7.2 Relative durations

By contrast, the `cal::relative_duration` type represents a “calendar” duration, like `1 month`. Because months have different number of days, `1 month` doesn’t correspond to a fixed number of milliseconds, but it’s often a useful quantity to represent recurring events, postponements, etc.

---

**Note:** The `cal::relative_duration` type supports the same units as `duration`, plus `days`, `weeks`, `months`, `years`, `decades`, `centuries`, and `millennia`.

---

To declare relative duration literals:

```
db> select <cal::relative_duration>'15 milliseconds';
{<cal::relative_duration>'PT.015S'}
db> select <cal::relative_duration>'2 months 3 weeks 45 minutes';
{<cal::relative_duration>'P2M21DT45M'}
db> select <cal::relative_duration>'-7 millennia';
{<cal::relative_duration>'P-7000Y'}
```

### 3.1.7.3 Date durations

The `cal::date_duration` represents spans consisting of some number of `months` and `days`. This type is primarily intended to simplify logic involving `cal::local_date` values.

```
db> select <cal::date_duration>'5 days';
{<cal::date_duration>'P5D'}
db> select <cal::local_date>'2022-06-25' + <cal::date_duration>'5 days';
{<cal::local_date>'2022-06-30'}
db> select <cal::local_date>'2022-06-30' - <cal::local_date>'2022-06-25';
{<cal::date_duration>'P5D'}
```

EdgeQL supports a set of functions and operators on duration types.

Comparison operators	<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>
Arithmetic	<code>dt + dt</code> <code>dt - dt</code>
Duration string parsing	<code>to_duration()</code> <code>cal::to_relative_duration()</code> <code>cal::to_date_duration()</code>
Component extraction	<code>duration_get()</code>
Conversion	<code>duration_truncate()</code> <code>cal::duration_normalize_hours()</code> <code>cal::duration_normalize_days()</code>

### 3.1.8 Ranges

Ranges represent a range of orderable scalar values. A range comprises a lower bound, upper bound, and two boolean flags indicating whether each bound is inclusive.

Create a range literal with the `range` constructor function.

```
db> select range(1, 10);
{range(1, 10, inc_lower := true, inc_upper := false)}
db> select range(2.2, 3.3);
{range(2.2, 3.3, inc_lower := true, inc_upper := false)}
```

Ranges can be *empty*, when the upper and lower bounds are equal.

```
db> select range(1, 1);
{range({}, empty := true)}
```

Ranges can be *unbounded*. An empty set is used to indicate the lack of a particular upper or lower bound.

```
db> select range(4, <int64>{});
{range(4, {})}
db> select range(<int64>{}, 4);
{range({}, 4)}
db> select range(<int64>{}, <int64>{});
{range({}, {})}
```

To compute the set of concrete values defined by a range literal, use `range_unpack`. An empty range will unpack to the empty set. Unbounded ranges cannot be unpacked.

```
db> select range_unpack(range(0, 10));
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
db> select range_unpack(range(1, 1));
{}
db> select range_unpack(range(0, <int64>{}));
edgedb error: InvalidValueError: cannot unpack an unbounded range
```

### 3.1.9 Bytes

The `bytes` type represents raw binary data.

```
db> select b'bina\x01ry';
{b'bina\x01ry'}
```

There is a special syntax for declaring “raw byte strings”. Raw byte strings treat the backslash \ as a literal character instead of an escape character.

```
db> select rb'hello\nthere';
{b'hello\nthere'}
db> select br'\';
{b'\\'}
```

### 3.1.10 Arrays

An array is an *ordered* collection of values of the *same type*. For example:

```
db> select [1, 2, 3];
[[1, 2, 3]]
db> select ['hello', 'world'];
[['hello', 'world']]
db> select [(1, 2), (100, 200)];
[(1, 2), (100, 200)]
```

EdgeQL provides a set of functions and operators on arrays.

Indexing and slicing	<code>array[i]</code> <code>array[from:to]</code> <code>array_get()</code>
Concatenation	<code>array ++ array</code>
Comparison operators	<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>
Utilities	<code>len()</code> <code>array_join()</code>
Search	<code>contains()</code> <code>find()</code>
Conversion to/from sets	<code>array_agg()</code> <code>array_unpack()</code>

See [Standard Library > Array](#) for a complete reference on array data types.

### 3.1.11 Tuples

A tuple is *fixed-length, ordered* collection of values, each of which may have a *different type*. The elements of a tuple can be of any type, including scalars, arrays, other tuples, and object types.

```
db> select ('Apple', 7, true);
{('Apple', 7, true)}
```

Optionally, you can assign a key to each element of a tuple. These are known as *named tuples*. You must assign keys to all or none of the elements; you can't mix-and-match.

```
db> select (fruit := 'Apple', quantity := 3.14, fresh := true);
{(fruit := 'Apple', quantity := 3.14, fresh := true)}
```

### 3.1.11.1 Indexing tuples

Tuple elements can be accessed with dot notation. Under the hood, there's no difference between named and unnamed tuples. Named tuples support key-based and numerical indexing.

```
db> select (1, 3.14, 'red').0;
{1}
db> select (1, 3.14, 'red').2;
{'red'}
db> select (name := 'george', age := 12).name;
{'george'}
db> select (name := 'george', age := 12).0;
{'george'}
```

---

**Important:** When you query an *unnamed tuple* using one of EdgeQL's *client libraries*, its value is converted to a list/array. When you fetch a *named tuple*, it is converted to an object/dictionary/hashmap.

---

For a full reference on tuples, see [Standard Library > Tuple](#).

### 3.1.12 JSON

The `json` scalar type is a stringified representation of structured data. JSON literals are declared by explicitly casting other values or passing a properly formatted JSON string into `to_json()`. Any type can be converted into JSON except `bytes`.

```
db> select <json>5;
{'5'}
db> select <json>"a string";
{"a string"}
db> select <json>["this", "is", "an", "array"];
[{"this", "is", "an", "array"}]
db> select <json>("unnamed tuple", 2);
[{"unnamed tuple", 2}]
db> select <json>(name := "named tuple", count := 2);
[{"name": "named tuple",
 "count": 2}]
db> select to_json('{"a": 2, "b": 5}');
[{"a": 2, "b": 5}]
```

JSON values support indexing operators. The resulting value is also of type `json`.

```
db> select to_json('{"a": 2, "b": 5}')['a'];
{2}
db> select to_json('["a", "b", "c"]')[2];
{"c"}
```

EdgeQL supports a set of functions and operators on `json` values. Refer to the [Standard Library > JSON](#) or click an item below for detailed documentation.

Indexing	<code>json[i]</code> <code>json[from:to]</code> <code>json[name]</code> <code>json_get()</code>
Merging	<code>json ++ json</code>
Comparison operators	<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>
Conversion to/from strings	<code>to_json()</code> <code>to_str()</code>
Conversion to/from sets	<code>json_array_unpack()</code> <code>json_object_unpack()</code>
Introspection	<code>json_typeof()</code>

## 3.2 Sets

### 3.2.1 Everything is a set

All values in EdgeQL are actually **sets**: a collection of values of a given **type**. All elements of a set must have the same type. The number of items in a set is known as its **cardinality**. A set with a cardinality of zero is referred to as an **empty set**. A set with a cardinality of one is known as a **singleton**.

### 3.2.2 Constructing sets

Set literals are declared with *set constructor* syntax: a comma-separated list of values inside a set of {curly braces}.

```
db> select {"set", "of", "strings"};
{"set", "of", "strings"}
db> select {1, 2, 3};
{1, 2, 3}
```

In actuality, curly braces are a syntactic sugar for the `union` operator. The previous examples are perfectly equivalent to the following:

```
db> select "set" union "of" union "strings";
{"set", "of", "strings"}
db> select 1 union 2 union 3;
{1, 2, 3}
```

A consequence of this is that nested sets are *flattened*.

```
db> select {1, {2, {3, 4}}};
{1, 2, 3, 4}
db> select 1 union (2 union (3 union 4));
{1, 2, 3, 4}
```

All values in a set must have the same type. For convenience, EdgeDB will *implicitly cast* values to other types, as long as there is no loss of information (e.g. converting a `int16` to an `int64`). For a full reference, see the casting table in [Standard Library > Casts](#).

```
db> select {1, 1.5};
{1.0, 1.5}
db> select {1, 1234.5678n};
{1.0n, 1234.5678n}
```

Attempting to declare a set containing elements of *incompatible* types is not permitted.

```
db> select {"apple", 3.14};
error: QueryError: set constructor has arguments of incompatible types
'std::str' and 'std::float64'
```

**Note:** Types are considered *compatible* if one can be implicitly cast into the other. For reference on implicit castability, see [Standard Library > Casts](#).

### 3.2.3 Literals are singletons

Literal syntax like 6 or "hello world" is just a shorthand for declaring a *singleton* of a given type. This is why the literals we created in the previous section were printed inside braces: to indicate that these values are *actually sets*.

```
db> select 6;
{6}
db> select "hello world";
>{"hello world"}
```

Wrapping a literal in curly braces does not change the meaning of the expression. For instance, "hello world" is *exactly equivalent* to {"hello world"}.

```
db> select {"hello world"};
>{"hello world"}
db> select "hello world" = {"hello world"};
{true}
```

You can retrieve the cardinality of a set with the `count()` function.

```
db> select count('aaa');
{1}
db> select count({'aaa', 'bbb'});
{2}
```

### 3.2.4 Empty sets

The reason EdgeQL introduced the concept of *sets* is to eliminate the concept of `null`. In SQL databases `null` is a special value denoting the absence of data; in EdgeDB the absence of data is just an empty set.

**Note:** Why is the existence of NULL a problem? Put simply, it's an edge case that permeates all of SQL and is often handled inconsistently in different circumstances. A number of specific inconsistencies are documented in detail in the [We Can Do Better Than SQL](#) post on the EdgeDB blog. For broader context, see Tony Hoare's talk “The Billion Dollar Mistake”.

Declaring empty sets isn't as simple as `{}`; in EdgeQL, all expressions are *strongly typed*, including empty sets. With nonempty sets (like `{1, 2, 3}`), the type is inferred from the set's contents (`int64`). But with empty sets this isn't possible, so an *explicit cast* is required.

```
db> select {};
error: QueryError: expression returns value of indeterminate type
```

(continues on next page)

(continued from previous page)

```
- query:1:8
1 | select {};
|   ^ Consider using an explicit type cast.

db> select <int64>{};
{ }

db> select <str>{};
{ }

db> select count(<str>{});
{0}
```

You can check whether or not a set is *empty* with the `exists` operator.

```
db> select exists <str>{};
{false}
db> select exists {'not', 'empty'};
{true}
```

### 3.2.5 Set references

A set reference is a *pointer* to a set of values. Most commonly, this is the name of an *object type* you've declared in your schema.

```
db> select User;
{
  default::User {id: 9d2ce01c-35e8-11ec-acc3-83b1377efea0},
  default::User {id: b0e0dd0c-35e8-11ec-acc3-abf1752973be},
}
db> select count(User);
{2}
```

It may also be an *alias*, which can be defined in a `with block` or as an *alias declaration* in your schema.

---

**Note:** In the example above, the `User` object type was declared inside the `default` module. If it was in a non-default module (say, `my_module`, we would need to use its *fully-qualified* name.

```
db> select my_module::User;
```

---

### 3.2.6 Multisets

Technically sets in EdgeDB are actually *multisets*, because they can contain duplicates of the same element. To eliminate duplicates, use the `distinct` set operator.

```
db> select {'aaa', 'aaa', 'aaa'};
{'aaa', 'aaa', 'aaa'}
db> select distinct {'aaa', 'aaa', 'aaa'};
{'aaa'}
```

### 3.2.7 Checking membership

Use the `in` operator to check whether a set contains a particular element.

```
db> select 'aaa' in {'aaa', 'bbb', 'ccc'};
{true}
db> select 'ddd' in {'aaa', 'bbb', 'ccc'};
{false}
```

### 3.2.8 Merging sets

Use the `union` operator to merge two sets.

```
db> select 'aaa' union 'bbb' union 'ccc';
{'aaa', 'bbb', 'ccc'}
db> select {1, 2} union {3.1, 4.4};
{1.0, 2.0, 3.1, 4.4}
```

### 3.2.9 Finding common members

Use the `intersect` operator to find common members between two sets.

```
db> select {1, 2, 3, 4, 5} intersect {3, 4, 5, 6, 7};
{3, 5, 4}
db> select {'a', 'b', 'c', 'd', 'e'} intersect {'c', 'd', 'e', 'f', 'g'};
{'e', 'd', 'c'}
```

If set members are repeated in both sets, they will be repeated in the set produced by `intersect` the same number of times they are repeated in both of the operand sets.

```
db> select {0, 1, 1, 1, 2, 3, 3} intersect {1, 3, 3, 3, 3, 3};
{1, 3, 3}
```

In this example, 1 appears three times in the first set but only once in the second, so it appears only once in the result. 3 appears twice in the first set and five times in the second. Both 3 appearances in the first set are overlapped by 3 appearances in the second, so they both end up in the resulting set.

### 3.2.10 Removing common members

Use the `except` operator to leave only the members in the first set that do not appear in the second set.

```
db> select {1, 2, 3, 4, 5} except {3, 4, 5, 6, 7};
{1, 2}
db> select {'a', 'b', 'c', 'd', 'e'} except {'c', 'd', 'e', 'f', 'g'};
{'b', 'a'}
```

When `except` eliminates a common member that is repeated, it never eliminates more than the number of instances of that member appearing in the second set.

```
db> select {0, 1, 1, 1, 2, 3, 3} except {1, 3, 3, 3, 3, 3};
{0, 1, 1, 2}
```

In this example, both sets share the member 1. The first set contains three of them while the second contains only one. The result retains two 1 members from the first set since the sets shared only a single 1 in common. The second set has five 3 members to the first set's two, so both of the first set's 3 members are eliminated from the resulting set.

### 3.2.11 Coalescing

Occasionally in queries, you need to handle the case where a set is empty. This can be achieved with a coalescing operator `??`. This is commonly used to provide default values for optional *query parameters*.

```
db> select 'value' ?? 'default';
{'value'}
db> select <str>{} ?? 'default';
{'default'}
```

---

**Note:** Coalescing is an example of a function/operator with *optional inputs*. By default, passing an empty set into a function/operator will “short circuit” the operation and return an empty set. However it’s possible to mark inputs as *optional*, in which case the operation will be defined over empty sets. Another example is `count()`, which returns `{0}` when an empty set is passed as input.

---

### 3.2.12 Inheritance

EdgeDB schemas support *inheritance*; types (usually object types) can extend one or more other types. For instance you may declare an abstract object type `Media` that is extended by `Movie` and `TVShow`.

```
abstract type Media {
    required property title -> str;
}

type Movie extending Media {
    property release_year -> int64;
}

type TVShow extending Media {
    property num_seasons -> int64;
}
```

```
abstract type Media {
    required title: str;
}

type Movie extending Media {
    release_year: int64;
}

type TVShow extending Media {
    num_seasons: int64;
}
```

A set of type `Media` may contain both `Movie` and `TVShow` objects.

```
db> select Media;
{
    default::Movie {id: 9d2ce01c-35e8-11ec-acc3-83b1377efea0},
    default::Movie {id: 3bfe4900-3743-11ec-90ee-cb73d2740820},
    default::TVShow {id: b0e0dd0c-35e8-11ec-acc3-abf1752973be},
}
```

We can use the *type intersection* operator [`is <type>`] to restrict the elements of a set by subtype.

```
db> select Media[is Movie];
{
    default::Movie {id: 9d2ce01c-35e8-11ec-acc3-83b1377efea0},
    default::Movie {id: 3bfe4900-3743-11ec-90ee-cb73d2740820},
}
db> select Media[is TVShow];
{
    default::TVShow {id: b0e0dd0c-35e8-11ec-acc3-abf1752973be}
}
```

Type filters are commonly used in conjunction with [backlinks](#).

### 3.2.13 Aggregate vs element-wise operations

EdgeQL provides a large library of built-in functions and operators for handling data structures. It's useful to consider functions/operators as either *aggregate* or *element-wise*.

---

**Note:** This is an over-simplification, but it's a useful mental model when just starting out with EdgeDB. For a more complete guide, see [Reference > Cardinality](#).

*Aggregate* operations are applied to the set *as a whole*; they accept a set with arbitrary cardinality and return a *singleton* (or perhaps an empty set if the input was also empty).

```
db> select count({'aaa', 'bbb'});
{2}
db> select sum({1, 2, 3});
{6}
db> select min({1, 2, 3});
{1}
```

Element-wise operations are applied on *each element* of a set.

```
db> select str_upper({'aaa', 'bbb'});
{'AAA', 'BBB'}
db> select {1, 2, 3} ^ 2;
{1, 4, 9}
db> select str_split("hello world", "hi again"), " ";
{["hello", "world"], ["hi", "again"]}
```

When an *element-wise* operation accepts two or more inputs, the operation is applied to all possible combinations of inputs; in other words, the operation is applied to the *Cartesian product* of the inputs.

```
db> select {'aaa', 'bbb'} ++ {'ccc', 'ddd'};
{'aaaccc', 'aaaddd', 'bbbccc', 'bbbddd'}
```

Accordingly, operations involving an empty set typically return an empty set. In contrast, aggregate operations like `count()` are able to operate on empty sets.

```
db> select <str>{} ++ 'ccc';
{}
db> select count(<str>{});
{0}
```

For a more complete discussion of cardinality, see [Reference > Cardinality](#).

### 3.2.14 Conversion to/from arrays

Both arrays and sets are collections of values that share a type. EdgeQL provides ways to convert one into the other.

---

**Note:** Remember that *all values* in EdgeQL are sets; an array literal is just a singleton set of arrays. So here, “converting” a set into an array means converting a set of type `x` into another set with cardinality 1 (a singleton) and type `array<x>`.

---

```
db> select array_unpack([1,2,3]);
{1, 2, 3}
db> select array_agg({1,2,3});
{[1, 2, 3]}
```

Arrays are an *ordered collection*, whereas sets are generally unordered (unless explicitly sorted with an `order by` clause in a `select` statement).

Element-wise scalar operations in the standard library cannot be applied to arrays, so sets of scalars are typically easier to manipulate, search, and transform than arrays.

```
db> select str_trim({' hello', 'world '});
{'hello', 'world'}
db> select str_trim([' hello', 'world ']);
error: QueryError: function "str_trim(arg0: array<std::string>)" does not exist
```

Most `aggregate` operations have analogs that operate on arrays. For instance, the set function `count()` is analogous to the array function `len()`.

### 3.2.15 Reference

Set operators	<a href="#">distinct in union exists if..else ?? detached [is type]</a>
Utility functions	<a href="#">count()</a> <a href="#">enumerate()</a>
Cardinality assertion	<a href="#">assert_distinct()</a> <a href="#">assert_single()</a> <a href="#">assert_exists()</a>

See also
<a href="#">Tutorial &gt; Building Blocks &gt; Sets</a>

## 3.3 Paths

A *path expression* (or simply a *path*) represents a set of values that are reachable by traversing a given sequence of links or properties from some source set of objects.

Consider the following schema:

```
type User {
    required property email -> str;
    multi link friends -> User;
}

type BlogPost {
    required property title -> str;
    required link author -> User;
}
```

```
type User {
    required email: str;
    multi friends: User;
}

type BlogPost {
    required title: str;
    required author: User;
}
```

The simplest path is simply `User`. This is a *set reference* that refers to all `User` objects in the database.

```
select User;
```

Paths can traverse links. The path below refers to *all Users who are the friend of another User*.

```
select User.friends;
```

Paths can traverse to an arbitrary depth in a series of nested links.

```
select BlogPost.author.friends.friends;
```

Paths can terminate with a property reference.

```
select BlogPost.title; # all blog post titles
select BlogPost.author.email; # all author emails
select User.friends.email; # all friends' emails
```

### 3.3.1 Backlinks

All examples thus far have traversed links in the *forward direction*, however it's also possible to traverse links *backwards* with `.<` notation. These are called **backlinks**.

Starting from each user, the path below traverses all *incoming* links labeled `author` and returns the union of their sources.

```
select User.<author;
```

As written, EdgeDB infers the *type* of this expression to be `BaseObject`, not `BlogPost`. Why? Because in theory, there may be several links named `author` that point to `User`.

---

**Note:** `BaseObject` is the root ancestor of all object types and it only contains a single property, `id`.

---

Consider the following addition to the schema:

```
type User {  
    # as before  
}  
  
type BlogPost {  
    required link author -> User;  
}  
  
+ type Comment {  
+    required link author -> User;  
+ }
```

```
type User {  
    # as before  
}  
  
type BlogPost {  
    required author: User;  
}  
  
+ type Comment {  
+    required author: User;  
+ }
```

With the above schema, the path `User.<author` would return a mixed set of `BlogPost` and `Comment` objects. This may be desirable in some cases, but commonly you'll want to narrow the results to a particular type. To do so, use the `type intersection` operator: `[is Foo]`:

```
select User.<author[is BlogPost]; # returns all blog posts  
select User.<author[is Comment]; # returns all comments
```

### 3.3.2 Link properties

Paths can also reference *link properties* with @ notation. To demonstrate this, let's add a property to the `User.friends` link:

```
type User {
    required property email -> str;
-   multi link friends -> User;
+   multi link friends -> User {
+     property since -> cal::local_date;
+   }
}
```

```
type User {
    required email: str;
-   multi friends: User;
+   multi friends: User {
+     since: cal::local_date;
+   }
}
```

The following represents a set of all dates on which friendships were formed.

```
select User.friends@since;
```

### 3.3.3 Path roots

For simplicity, all examples above use set references like `User` as the root of the path; however, the root can be *any expression* returning object types. Below, the root of the path is a *subquery*.

```
db> with edgedb_lovers := (
...   select BlogPost filter .title ilike "EdgeDB is awesome"
... )
... select edgedb_lovers.author;
```

This expression returns a set of all `Users` who have written a blog post titled “EdgeDB is awesome”.

For a full syntax definition, see the [Reference > Paths](#).

## 3.4 Types

The foundation of EdgeQL is EdgeDB’s rigorous type system. There is a set of EdgeQL operators and functions for changing, introspecting, and filtering by types.

### 3.4.1 Type expressions

Type expressions are exactly what they sound like: EdgeQL expressions that refer to a type. Most commonly, these are simply the *names* of established types: `str`, `int64`, `BlogPost`, etc. Arrays and tuples have a dedicated type syntax.

Type	Syntax
Array	<code>array&lt;x&gt;</code>
Tuple (unnamed)	<code>tuple&lt;x, y, z&gt;</code>
Tuple (named)	<code>tuple&lt;foo: x, bar: y&gt;</code>

For additional details on type syntax, see [Schema > Primitive Types](#).

### 3.4.2 Type casting

Type casting is used to convert primitive values into another type. Casts are indicated with angle brackets containing a type expression.

```
db> select <str>10;
{"10"}
db> select <bigint>10;
{10n}
db> select <array<str>>[1, 2, 3];
[['1', '2', '3']]
db> select <tuple<str, float64, bigint>>(1, 2, 3);
{'1', 2, 3n}
```

Type casts are useful for declaring literals for types like `datetime`, `uuid`, and `int16` that don't have a dedicated syntax.

```
db> select <datetime>'1999-03-31T15:17:00Z';
{<datetime>'1999-03-31T15:17:00Z'}
db> select <int16>42;
{42}
db> select <uuid>'89381587-705d-458f-b837-860822e1b219';
{89381587-705d-458f-b837-860822e1b219}
```

There are limits to what values can be cast to a certain type. In some cases two types are entirely incompatible, like `bool` and `int64`; in other cases, the source data must be in a particular format, like casting `str` to `datetime`. For a comprehensive table of castability, see [Standard Library > Casts](#).

Type casts can only be used on primitive expressions, not object type expressions. Every object stored in the database is strongly and immutably typed; you can't simply convert an object to an object of a different type.

```
db> select <BlogPost>10;
QueryError: cannot cast 'std::int64' to 'default::BlogPost'
db> select <int64>'asdf';
InvalidValueError: invalid input syntax for type std::int64: "asdf"
db> select <int16>100000000000000n;
NumericOutOfRangeError: std::int16 out of range
```

### 3.4.3 Type intersections

All elements of a given set have the same type; however, in the context of *sets of objects*, this type might be abstract and contain elements of multiple concrete subtypes. For instance, a set of `Media` objects may contain both `Movie` and `TVShow` objects.

```
db> select Media;
{
    default::Movie {id: 9d2ce01c-35e8-11ec-acc3-83b1377efea0},
    default::Movie {id: 3bfe4900-3743-11ec-90ee-cb73d2740820},
    default::TVShow {id: b0e0dd0c-35e8-11ec-acc3-abf1752973be},
}
```

We can use the *type intersection* operator to restrict the elements of a set by subtype.

```
db> select Media[is Movie];
{
    default::Movie {id: 9d2ce01c-35e8-11ec-acc3-83b1377efea0},
    default::Movie {id: 3bfe4900-3743-11ec-90ee-cb73d2740820},
}
```

Logically, this computes the intersection of the `Media` and `Movie` sets; since only `Movie` objects occur in both sets, this can be conceptualized as a “filter” that removes all elements that aren’t of type `Movie`.

### 3.4.4 Type checking

The `[is foo]` “type intersection” syntax should not be confused with the *type checking* operator `is`.

```
db> select 5 is int64;
{true}
db> select {3.14, 2.718} is not int64;
{true, true}
db> select Media is Movie;
{true, true, false}
```

### 3.4.5 The `typeof` operator

The type of any expression can be extracted with the `typeof` operator. This can be used in any expression that expects a type.

```
db> select <typeof 5>'100';
{100}
db> select "tuna" is typeof "trout";
{true}
```

### 3.4.6 Introspection

The entire type system of EdgeDB is *stored inside EdgeDB*. All types are introspectable as instances of the `schema::Type` type. For a set of introspection examples, see [Guides > Introspection](#). To try introspection for yourself, see our interactive introspection tutorial.

## 3.5 Parameters

### edb-alt-title Query Parameters

EdgeQL queries can reference parameters with \$ notation. The value of these parameters are supplied externally.

```
select <str>$var;
select <int64>$a + <int64>$b;
select BlogPost filter .id = <uuid>$blog_id;
```

Note that we provided an explicit type cast before the parameter. This is required, as it enables EdgeDB to enforce the provided types at runtime.

Parameters can be named or unnamed tuples.

```
select <tuple<str, bool>>$var;
select <optional tuple<str, bool>>$var;
select <tuple<name: str, flag: bool>>$var;
select <optional tuple<name: str, flag: bool>>$var;
```

### 3.5.1 Usage with clients

#### 3.5.1.1 REPL

When you include a parameter reference in an EdgeDB REPL, you'll be prompted interactively to provide a value or values.

```
db> select 'I ' ++ <str>$var ++ '!';
Parameter <str>$var: EdgeDB
{'I  EdgeDB!'}
```

#### 3.5.1.2 Python

```
await client.query(
    "select 'I ' ++ <str>$var ++ '!';
    var='lamp')"

await client.query(
    "select <datetime>$date;",
    date=datetime.today())
```

### 3.5.1.3 JavaScript

```
await client.query("select 'I ' ++ <str>$name ++ '!';", {
  name: "rock and roll"
});

await client.query("select <datetime>$date;", {
  date: new Date()
});
```

### 3.5.1.4 Go

```
var result string
err = db.QuerySingle(ctx,
    `select 'I ' ++ <str>$var ++ '!';`,
    &result, "Golang")

var date time.Time
err = db.QuerySingle(ctx,
    `select <datetime>$date;`,
    &date, time.Now())
```

Refer to the Datatypes page of your preferred [client library](#) to learn more about mapping between EdgeDB types and language-native types.

## 3.5.2 Parameter types and JSON

Prior to EdgeDB 3.0, parameters can be only [scalars](#) or arrays of scalars. In EdgeDB 3.0, parameters can also be tuples. This may seem limiting at first, but in actuality this doesn't impose any practical limitation on what can be parameterized. To pass complex structures as parameters, use EdgeDB's built-in [JSON](#) functionality.

```
db> with data := <json>$data
... insert Movie {
...   title := <str>data['title'],
...   release_year := <int64>data['release_year'],
... };
Parameter <json>$data: {"title": "The Marvels", "release_year": 2023}
{default::Movie {id: 8d286cfe-3c0a-11ec-aa68-3f3076ebd97f}}
```

Arrays can be “unpacked” into sets and assigned to `multi` links or properties.

```
with friends := (
  select User filter .id in array_unpack(<array<uuid>>$friend_ids)
)
insert User {
  name := <str>$name,
  friends := friends,
};
```

### 3.5.3 Optional parameters

By default, query parameters are required; the query will fail if the parameter value is an empty set. You can use an optional modifier inside the type cast if the parameter is optional.

```
db> select <optional str>$name;
Parameter <str>$name (Ctrl+D for empty set `{}`):
{}
```

When using a client library, pass the idiomatic null pointer for your language: `null`, `None`, `nil`, etc.

---

**Note:** The `<required foo>` type cast is also valid (though redundant) syntax.

```
select <required str>$name;
```

---

### 3.5.4 What can be parameterized?

Any data manipulation language (DML) statement can be parameterized: `select`, `insert`, `update`, and `delete`.

Schema definition language (SDL) and `configure` statements **cannot** be parameterized. Data definition language (DDL) has limited support for parameters, but it's not a recommended pattern. Some of the limitations might be lifted in the future versions.

## 3.6 Select

The `select` command retrieves or computes a set of values. We've already seen simple queries that select primitive values.

```
db> select 'hello world';
{'hello world'}
db> select [1, 2, 3];
{[1, 2, 3]}
db> select {1, 2, 3};
{1, 2, 3}
```

With the help of a `with` block, we can add filters, ordering, and pagination clauses.

```
db> with x := {1, 2, 3, 4, 5}
... select x
... filter x >= 3;
{3, 4, 5}
db> with x := {1, 2, 3, 4, 5}
... select x
... order by x desc;
{5, 4, 3, 2, 1}
db> with x := {1, 2, 3, 4, 5}
... select x
... offset 1 limit 3;
{2, 3, 4}
```

These queries can also be rewritten to use inline aliases, like so:

```
db> select x := {1, 2, 3, 4, 5}
... filter x >= 3;
```

### 3.6.1 Selecting objects

However most queries are selecting *objects* that live in the database. For demonstration purposes, the queries below assume the following schema.

```
module default {
    abstract type Person {
        required property name -> str { constraint exclusive };
    }

    type Hero extending Person {
        property secret_identity -> str;
        multi link villains := .<nemesis[is Villain];
    }

    type Villain extending Person {
        link nemesis -> Hero;
    }

    type Movie {
        required property title -> str { constraint exclusive };
        required property release_year -> int64;
        multi link characters -> Person;
    }
}
```

```
module default {
    abstract type Person {
        required name: str { constraint exclusive };
    }

    type Hero extending Person {
        secret_identity: str;
        multi link villains := .<nemesis[is Villain];
    }

    type Villain extending Person {
        nemesis: Hero;
    }

    type Movie {
        required title: str { constraint exclusive };
        required release_year: int64;
        multi characters: Person;
    }
}
```

Let's start by selecting all `Villain` objects in the database. In this example, there are only three. Remember, `Villain` is a [reference](#) to the set of all `Villain` objects.

```
db> select Villain;
{
    default::Villain {id: ea7bad4c...},
    default::Villain {id: 6ddbb04a...},
    default::Villain {id: b233ca98...},
}
```

---

**Note:** For the sake of readability, the `id` values have been truncated.

---

By default, this only returns the `id` of each object. If serialized to JSON, this result would look like this:

```
[{"id": "ea7bad4c-35d6-11ec-9519-0361f8abd380"}, {"id": "6ddbb04a-3c23-11ec-b81f-7b7516f2a868"}, {"id": "b233ca98-3c23-11ec-b81f-6ba8c4f0084e"}]
```

Learn to select objects by trying it in [our interactive object query tutorial](#).

## 3.6.2 Shapes

To specify which properties to select, we attach a **shape** to `Villain`. A shape can be attached to any object type expression in EdgeQL.

```
db> select Villain { id, name };
{
    default::Villain { id: ea7bad4c..., name: 'Whiplash' },
    default::Villain { id: 6ddbb04a..., name: 'Green Goblin' },
    default::Villain { id: b233ca98..., name: 'Doc Ock' },
}
```

To learn to use shapes by trying them yourself, see [our interactive shapes tutorial](#).

### 3.6.2.1 Nested shapes

Nested shapes can be used to fetch linked objects and their properties. Here we fetch all `Villain` objects and their nemeses.

```
db> select Villain {
...     name,
...     nemesis: { name }
... };
{
    default::Villain {
        name: 'Green Goblin',
        nemesis: default::Hero {name: 'Spider-Man'},
    },
    ...
}
```

In the context of EdgeQL, computed links like `Hero.villains` are treated identically to concrete/non-computed links like `Villain.nemesis`.

```
db> select Hero {
...   name,
...   villains: { name }
... };
{
  default::Hero {
    name: 'Spider-Man',
    villains: {
      default::Villain {name: 'Green Goblin'},
      default::Villain {name: 'Doc Ock'},
    },
  },
  ...
}
```

### 3.6.2.2 Splats

Splats allow you to select all properties of a type using the asterisk (\*) or all properties of the type and a single level of linked types with a double asterisk (\*\*).

If you have this schema:

```
module default {
  abstract type Person {
    required name: str { constraint exclusive };
  }

  type Hero extending Person {
    secret_identity: str;
    multi link villains := .<nemesis[is Villain];
  }

  type Villain extending Person {
    nemesis: Hero;
  }

  type Movie {
    required title: str { constraint exclusive };
    required release_year: int64;
    multi characters: Person;
  }
}
```

splats will help you more easily select all properties when using the REPL. You can select all of an object's properties using the single splat:

```
db> select Movie {*};
{
  default::Movie {
    id: 6fe5c3ec-b776-11ed-8bef-3b2fba99fe8a,
```

(continues on next page)

(continued from previous page)

```

    release_year: 2021,
    title: 'Spiderman: No Way Home',
},
default::Movie {
    id: 76998656-b776-11ed-8bef-237907a987fa,
    release_year: 2008,
    title: 'Iron Man'
},
}

```

or you can select all of an object's properties and the properties of a single level of nested objects with the double splat:

```

db> select Movie {**};
{
    default::Movie {
        id: 6fe5c3ec-b776-11ed-8bef-3b2fba99fe8a,
        release_year: 2021,
        title: 'Spiderman: No Way Home',
        characters: {
            default::Hero {
                id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7,
                name: 'Spiderman'
            },
            default::Villain {
                id: efa2c4bc-b777-11ed-99eb-43f835d79384,
                name: 'Electro'
            },
            default::Villain {
                id: f2c99a96-b775-11ed-8f7e-0b4a4a8e433e,
                name: 'Green Goblin'
            },
            default::Villain {
                id: f8ca354a-b775-11ed-8bef-273145019e1d,
                name: 'Doc Ock'
            },
        },
        default::Movie {
            id: 76998656-b776-11ed-8bef-237907a987fa,
            release_year: 2008,
            title: 'Iron Man',
            characters: {
                default::Hero {
                    id: 48edcf8c-b776-11ed-8bef-c7d61b6780d2,
                    name: 'Iron Man'
                },
                default::Villain {
                    id: 335f4104-b777-11ed-81eb-ab4de34e9c36,
                    name: 'Obadiah Stane'
                },
            },
        },
    },
}

```

(continues on next page)

(continued from previous page)

}

**Note:** Splits are not yet supported in function bodies.

The splat expands all properties defined on the type as well as inherited properties. Given the same schema shown above:

```
db> select Hero {*};
{
  default::Hero {
    id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7,
    name: 'Spiderman',
    secret_identity: 'Peter Parker'
  },
  default::Hero {
    id: 48edcf8c-b776-11ed-8bef-c7d61b6780d2,
    name: 'Iron Man',
    secret_identity: 'Tony Stark'
  },
}
```

The splat here expands the heroes' names even though the `name` property is not defined on the `Hero` type but on the `Person` type it extends. If we want to select heroes but get only properties defined on the `Person` type, we can do this instead:

```
db> select Hero {Person.*};
{
  default::Hero {
    id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7,
    name: 'Spiderman'
  },
  default::Hero {
    id: 48edcf8c-b776-11ed-8bef-c7d61b6780d2,
    name: 'Iron Man'
  },
}
```

If there are links on our `Person` type, we can use `Person.**` in a similar fashion to get all properties and one level of linked object properties, but only for links and properties that are defined on the `Person` type.

You can use the splat to expand properties using a *type intersection*. Maybe we want to select all `Person` objects with their names but also get any properties defined on the `Hero` for those `Person` objects which are also `Hero` objects:

```
db> select Person {
...   name,
...   [is Hero].*
... };
{
  default::Hero {
    id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7,
    name: 'Spiderman',
    secret_identity: 'Peter Parker'
```

(continues on next page)

(continued from previous page)

```

},
default::Hero {
    id: 48edcf8c-b776-11ed-8bef-c7d61b6780d2,
    name: 'Iron Man'
    secret_identity: 'Tony Stark'
},
default::Villain {
    id: efa2c4bc-b777-11ed-99eb-43f835d79384,
    name: 'Electro'
},
default::Villain {
    id: f2c99a96-b775-11ed-8f7e-0b4a4a8e433e,
    name: 'Green Goblin'
},
default::Villain {
    id: f8ca354a-b775-11ed-8bef-273145019e1d,
    name: 'Doc Ock'
},
default::Villain {
    id: 335f4104-b777-11ed-81eb-ab4de34e9c36,
    name: 'Obadiah Stane'
},
}
}

```

The double splat also works with type intersection expansion to expand both properties and links on the specified type.

```

db> select Person {
...     name,
...     [is Hero].**
... };
{
    default::Hero {
        id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7,
        name: 'Spiderman'
        secret_identity: 'Peter Parker',
        villains: {
            default::Villain {
                id: efa2c4bc-b777-11ed-99eb-43f835d79384,
                name: 'Electro'
            },
            default::Villain {
                id: f2c99a96-b775-11ed-8f7e-0b4a4a8e433e,
                name: 'Green Goblin'
            },
            default::Villain {
                id: f8ca354a-b775-11ed-8bef-273145019e1d,
                name: 'Doc Ock'
            }
        }
    },
    default::Hero {
        id: 48edcf8c-b776-11ed-8bef-c7d61b6780d2,
    }
}

```

(continues on next page)

(continued from previous page)

```

name: 'Iron Man'
secret_identity: 'Tony Stark'
villains: {
    default::Villain {
        id: 335f4104-b777-11ed-81eb-ab4de34e9c36,
        name: 'Obadiah Stane'
    }
},
default::Villain {
    id: efa2c4bc-b777-11ed-99eb-43f835d79384,
    name: 'Electro'
},
default::Villain {
    id: f2c99a96-b775-11ed-8f7e-0b4a4a8e433e,
    name: 'Green Goblin'
},
default::Villain {
    id: f8ca354a-b775-11ed-8bef-273145019e1d,
    name: 'Doc Ock'
},
default::Villain {
    id: 335f4104-b777-11ed-81eb-ab4de34e9c36,
    name: 'Obadiah Stane'
},
}

```

With this query, we get `name` for each `Person` and all the properties and one level of links on the `Hero` objects. We don't get `Villain` objects' nemeses because that link is not covered by our double splat which only expands `Hero` links. If the `Villain` type had properties defined on it, we wouldn't get those with this query either.

### 3.6.3 Filtering

To filter the set of selected objects, use a `filter <expr>` clause. The `<expr>` that follows the `filter` keyword can be *any boolean expression*.

To reference the `name` property of the `Villain` objects being selected, we use `Villain.name`.

```

db> select Villain {id, name}
... filter Villain.name = "Doc Ock";
{default::Villain {id: b233ca98..., name: 'Doc Ock'}}

```

---

**Note:** This query contains two occurrences of `Villain`. The first (outer) is passed as the argument to `select` and refers to the set of all `Villain` objects. However the *inner* occurrence is inside the *scope* of the `select` statement and refers to the *object being selected*.

However, this looks a little clunky, so EdgeQL provides a shorthand: just drop `Villain` entirely and simply use `.name`. Since we are selecting a set of `Villains`, it's clear from context that `.name` must refer to a link/property of the `Villain` type. In other words, we are in the *scope* of the `Villain` type.

```
db> select Villain {name}
... filter .name = "Doc Ock";
{default::Villain {name: 'Doc Ock'}}
```

Learn to filter your queries by trying it in [our interactive filters tutorial](#).

### 3.6.3.1 Filtering by ID

To filter by `id`, remember to cast the desired ID to `uuid`:

```
db> select Villain {id, name}
... filter .id = <uuid>"b233ca98-3c23-11ec-b81f-6ba8c4f0084e";
{
  default::Villain {
    id: 'b233ca98-3c23-11ec-b81f-6ba8c4f0084e',
    name: 'Doc Ock'
  }
}
```

### 3.6.3.2 Nested filters

Filters can be added at every level of shape nesting. The query below applies a filter to both the selected `Hero` objects and their linked `villains`.

```
db> select Hero {
...   name,
...   villains: {
...     name
...     } filter .name ilike "%er"
...   } filter .name ilike "%man";
{
  default::Hero {
    name: 'Iron Man',
    villains: {default::Villain {name: 'Justin Hammer'}},
  },
  default::Hero {
    name: 'Spider-Man',
    villains: {
      default::Villain {name: 'Shocker'},
      default::Villain {name: 'Tinkerer'},
      default::Villain {name: 'Kraven the Hunter'},
    },
  },
}
```

Note that the `scope` changes inside nested shapes. When we use `.name` in the outer `filter`, it refers to the name of the hero. But when we use `.name` in the nested `villains` shape, the scope has changed to `Villain`.

### 3.6.4 Ordering

Order the result of a query with an `order by` clause.

```
db> select Villain { name }
... order by .name;
{
    default::Villain {name: 'Abomination'},
    default::Villain {name: 'Doc Ock'},
    default::Villain {name: 'Green Goblin'},
    default::Villain {name: 'Justin Hammer'},
    default::Villain {name: 'Kraven the Hunter'},
    default::Villain {name: 'Loki'},
    default::Villain {name: 'Shocker'},
    default::Villain {name: 'The Vulture'},
    default::Villain {name: 'Tinkerer'},
    default::Villain {name: 'Zemo'},
}
```

The expression provided to `order by` may be *any* singleton expression, primitive or otherwise.

---

**Note:** In EdgeDB all values are orderable. Objects are compared using their `id`; tuples and arrays are compared element-by-element from left to right. By extension, the generic comparison operators `=`, `<`, `>`, etc. can be used with any two expressions of the same type.

---

You can also order by multiple expressions and specify the *direction* with an `asc` (default) or `desc` modifier.

---

**Note:** When ordering by multiple expressions, arrays, or tuples, the leftmost expression/element is compared. If these elements are the same, the next element is used to “break the tie”, and so on. If all elements are the same, the order is not well defined.

---

```
db> select Movie { title, release_year }
... order by
...     .release_year desc then
...     str_trim(.title) desc;
{
    default::Movie {title: 'Spider-Man: No Way Home', release_year: 2021},
    ...
    default::Movie {title: 'Iron Man', release_year: 2008},
}
```

When ordering by multiple expressions, each expression is separated with the `then` keyword. For a full reference on ordering, including how empty values are handled, see [Reference > Commands > Select](#).

### 3.6.5 Pagination

EdgeDB supports `limit` and `offset` clauses. These are typically used in conjunction with `order by` to maintain a consistent ordering across pagination queries.

```
db> select Villain { name }
...   order by .name
...   offset 3
...   limit 3;
{
    default::Villain {name: 'Hela'},
    default::Villain {name: 'Justin Hammer'},
    default::Villain {name: 'Kraven the Hunter'},
}
```

The expressions passed to `limit` and `offset` can be any singleton `int64` expression. This query fetches all Villains except the last (sorted by name).

```
db> select Villain {name}
...   order by .name
...   limit count(Villain) - 1;
{
    default::Villain {name: 'Abomination'},
    default::Villain {name: 'Doc Ock'},
    ...
    default::Villain {name: 'Winter Soldier'}, # no Zemo
}
```

You may pass the empty set to `limit` or `offset`. Passing the empty set is effectively the same as excluding `limit` or `offset` from your query (i.e., no limit or no offset). This is useful if you need to parameterize `limit` and/or `offset` but may still need to execute your query without providing one or the other.

```
db> select Villain {name}
...   order by .name
...   offset <optional int64>$offset
...   limit <optional int64>$limit;
Parameter <int64>$offset (Ctrl+D for empty set `{}`):
Parameter <int64>$limit (Ctrl+D for empty set `{}`):
{
    default::Villain {name: 'Abomination'},
    default::Villain {name: 'Doc Ock'},
    ...
}
```

---

**Note:** If you parameterize `limit` and `offset` and want to reserve the option to pass the empty set, make sure those parameters are optional as shown in the example above.

---

### 3.6.6 Computed fields

Shapes can contain *computed fields*. These are EdgeQL expressions that are computed on the fly during the execution of the query. As with other clauses, we can use *leading dot notation* (e.g. `.name`) to refer to the properties and links of the object type currently *in scope*.

```
db> select Villain {
...     name,
...     name_upper := str_upper(.name)
... };
{
    default::Villain {
        id: 4114dd56....,
        name: 'Abomination',
        name_upper: 'ABOMINATION',
    },
    ...
}
```

As with nested filters, the *current scope* changes inside nested shapes.

```
db> select Villain {
...     id,
...     name,
...     name_upper := str_upper(.name),
...     nemesis: {
...         secret_identity,
...         real_name_upper := str_upper(.secret_identity)
...     }
... };
{
    default::Villain {
        id: 6ddb04a....,
        name: 'Green Goblin',
        name_upper: 'GREEN GOBLIN',
        nemesis: default::Hero {
            secret_identity: 'Peter Parker',
            real_name_upper: 'PETER PARKER',
        },
    },
    ...
}
```

### 3.6.7 Backlinks

Fetching backlinks is a common use case for computed fields. To demonstrate this, let's fetch a list of all movies starring a particular Hero.

```
db> select Hero {
...     name,
...     movies := .<characters[is Movie] { title }
... } filter .name = "Iron Man";
```

(continues on next page)

(continued from previous page)

```
{
    default::Hero {
        name: 'Iron Man',
        movies: {
            default::Movie {title: 'Iron Man'},
            default::Movie {title: 'Iron Man 2'},
            default::Movie {title: 'Iron Man 3'},
            default::Movie {title: 'Captain America: Civil War'},
            default::Movie {title: 'The Avengers'},
        },
    },
}
```

**Note:** The computed backlink `movies` is a combination of the *backlink operator* `.<` and a type intersection `[is Movie]`. For a full reference on backlink syntax, see [EdgeQL > Paths](#).

Instead of re-declaring backlinks inside every query where they’re needed, it’s common to add them directly into your schema as computed links.

```
abstract type Person {
    required property name -> str {
        constraint exclusive;
    };
+   multi link movies := .<characters[is Movie]
}
```

```
abstract type Person {
    required name: str {
        constraint exclusive;
    };
+   multi link movies := .<characters[is Movie]
}
```

**Note:** In the example above, the `Person.movies` is a `multi link`. Including these keywords is optional, since EdgeDB can infer this from the assigned expression `.<characters[is Movie]`. However, it’s a good practice to include the explicit keywords to make the schema more readable and “sanity check” the cardinality.

This simplifies future queries; `Person.movies` can now be traversed in shapes just like a non-computed link.

```
select Hero {
    name,
    movies: { title }
} filter .name = "Iron Man";
```

### 3.6.8 Subqueries

There's no limit to the complexity of computed expressions. EdgeQL is designed to be fully composable; entire queries can be embedded inside each other. Below, we use a subquery to select all movies containing a villain's nemesis.

```
db> select Villain {
...   name,
...   nemesis_name := .nemesis.name,
...   movies_with_nemesis := (
...     select Movie { title }
...     filter Villain.nemesis in .characters
...   )
... };
{
  default::Villain {
    name: 'Loki',
    nemesis_name: 'Thor',
    movies_with_nemesis: {
      default::Movie {title: 'Thor'},
      default::Movie {title: 'Thor: The Dark World'},
      default::Movie {title: 'Thor: Ragnarok'},
      default::Movie {title: 'The Avengers'},
    },
    ...
  }
}
```

### 3.6.9 Polymorphic queries

**index** poly polymorphism nested shapes

All queries thus far have referenced concrete object types: `Hero` and `Villain`. However, both of these types extend the abstract type `Person`, from which they inherit the `name` property.

To learn how to leverage polymorphism in your queries, see [our interactive polymorphism tutorial](#).

#### 3.6.9.1 Polymorphic sets

It's possible to directly query all `Person` objects; the resulting set will be a mix of `Hero` and `Villain` objects (and possibly other subtypes of `Person`, should they be declared).

```
db> select Person { name };
{
  default::Villain {name: 'Abomination'},
  default::Villain {name: 'Zemo'},
  default::Hero {name: 'The Hulk'},
  default::Hero {name: 'Iron Man'},
  ...
}
```

You may also encounter such “mixed sets” when querying a link that points to an abstract type (such as `Movie.characters`) or a [union type](#).

```
db> select Movie {
...     title,
...     characters: {
...         name
...     }
... }
... filter .title = "Iron Man 2";
{
    default::Movie {
        title: 'Iron Man 2',
        characters: {
            default::Villain {name: 'Whiplash'},
            default::Villain {name: 'Justin Hammer'},
            default::Hero {name: 'Iron Man'},
            default::Hero {name: 'Black Widow'},
        },
    },
}
```

### 3.6.9.2 Polymorphic fields

We can fetch different properties *conditional* on the subtype of each object by prefixing property/link references with `[is <type>]`. This is known as a **polymorphic query**.

```
db> select Person {
...     name,
...     secret_identity := [is Hero].secret_identity,
...     number_of_villains := count([is Hero].villains),
...     nemesis := [is Villain].nemesis {
...         name
...     }
... };
{
    default::Villain {
        name: 'Green Goblin',
        secret_identity: {},
        number_of_villains: 0,
        nemesis: default::Hero {name: 'Spider-Man'},
    },
    default::Hero {
        name: 'Spider-Man',
        secret_identity: 'Peter Parker',
        number_of_villains: 6,
        nemesis: {},
    },
    ...
}
```

This syntax might look familiar; it's the *type intersection* again. In effect, this operator conditionally returns the value of the referenced field only if the object matches a particular type. If the match fails, an empty set is returned.

The line `secret_identity := [is Hero].secret_identity` is a bit redundant, since the computed property has the same name as the polymorphic field. In these cases, EdgeQL supports a shorthand.

```
db> select Person {
...     name,
...     [is Hero].secret_identity,
...     [is Villain].nemesis: {
...         name
...     }
... };
{
    default::Villain {
        name: 'Green Goblin',
        secret_identity: {},
        nemesis: default::Hero {name: 'Spider-Man'},
    },
    default::Hero {
        name: 'Spider-Man',
        secret_identity: 'Peter Parker',
        nemesis: {},
    },
    ...
}
```

### 3.6.9.3 Filtering polymorphic links

Relatedly, it's possible to filter polymorphic links by subtype. Below, we exclusively fetch the `Movie.characters` of type `Hero`.

```
db> select Movie {
...     title,
...     characters[is Hero]: {
...         secret_identity
...     },
... };
{
    default::Movie {
        title: 'Spider-Man: Homecoming',
        characters: {default::Hero {secret_identity: 'Peter Parker'}},
    },
    default::Movie {
        title: 'Iron Man',
        characters: {default::Hero {secret_identity: 'Tony Stark'}},
    },
    ...
}
```

### 3.6.10 Free objects

To select several values simultaneously, you can “bundle” them into a “free object”. Free objects are a set of key-value pairs that can contain any expression. Here, the term “free” is used to indicate that the object in question is not an instance of a particular *object type*; instead, it’s constructed ad hoc inside the query.

```
db> select {
...   my_string := "This is a string",
...   my_number := 42,
...   several_numbers := {1, 2, 3},
...   all_heroes := Hero { name }
... };
{
  {
    my_string: 'This is a string',
    my_number: 42,
    several_numbers: {1, 2, 3},
    all_heroes: {
      default::Hero {name: 'The Hulk'},
      default::Hero {name: 'Iron Man'},
      default::Hero {name: 'Spider-Man'},
      default::Hero {name: 'Thor'},
      default::Hero {name: 'Captain America'},
      default::Hero {name: 'Black Widow'},
    },
  },
}
```

Note that the result is a *singleton* but each key corresponds to a set of values, which may have any cardinality.

### 3.6.11 With block

All top-level EdgeQL statements (`select`, `insert`, `update`, and `delete`) can be prefixed with a `with` block. These blocks let you declare standalone expressions that can be used in your query.

```
db> with hero_name := "Iron Man"
... select Hero { secret_identity }
... filter .name = hero_name;
{default::Hero {secret_identity: 'Tony Stark'}}
```

For full documentation on `with`, see [EdgeQL > With](#).

See also
<a href="#">Reference &gt; Commands &gt; Select</a>
<a href="#">Cheatsheets &gt; Selecting data</a>
<a href="#">Tutorial &gt; Basic Queries &gt; Objects</a>
<a href="#">Tutorial &gt; Basic Queries &gt; Filters</a>
<a href="#">Tutorial &gt; Basic Queries &gt; Aggregates</a>
<a href="#">Tutorial &gt; Nested Structures &gt; Shapes</a>
<a href="#">Tutorial &gt; Nested Structures &gt; Polymorphism</a>

## 3.7 Insert

The `insert` command is used to create instances of object types. The code samples on this page assume the following schema:

```
module default {
    abstract type Person {
        required property name -> str { constraint exclusive };
    }

    type Hero extending Person {
        property secret_identity -> str;
        multi link villains := .<nemesis[is Villain];
    }

    type Villain extending Person {
        link nemesis -> Hero;
    }

    type Movie {
        required property title -> str { constraint exclusive };
        required property release_year -> int64;
        multi link characters -> Person;
    }
}
```

```
module default {
    abstract type Person {
        required name: str { constraint exclusive };
    }

    type Hero extending Person {
        secret_identity: str;
        multi link villains := .<nemesis[is Villain];
    }

    type Villain extending Person {
        nemesis: Hero;
    }

    type Movie {
        required title: str { constraint exclusive };
        required release_year: int64;
        multi characters: Person;
    }
}
```

### 3.7.1 Basic usage

You can `insert` instances of any *non-abstract* object type.

```
db> insert Hero {  
...   name := "Spider-Man",  
...   secret_identity := "Peter Parker"  
... };  
{default::Hero {id: b0fbe9de-3e90-11ec-8c12-ffa2d5f0176a}}
```

Similar to `selecting fields` in `select`, `insert` statements include a `shape` specified with curly braces; the values of properties/links are assigned with the `:=` operator.

Optional links or properties can be omitted entirely, as well as those with a `default` value (like `id`).

```
db> insert Hero {  
...   name := "Spider-Man"  
...   # secret_identity is omitted  
... };  
{default::Hero {id: b0fbe9de-3e90-11ec-8c12-ffa2d5f0176a}}
```

You can only `insert` instances of concrete (non-abstract) object types.

```
db> insert Person {  
...   name := "The Man With No Name"  
... };  
error: QueryError: cannot insert into abstract object type 'default::Person'
```

### 3.7.2 Inserting links

EdgeQL's composable syntax makes link insertion painless. Below, we insert “Spider-Man: No Way Home” and include all known heroes and villains as `characters` (which is basically true).

```
db> insert Movie {  
...   title := "Spider-Man: No Way Home",  
...   characters := (  
...     select Person  
...       filter .name in {  
...         'Spider-Man',  
...         'Doctor Strange',  
...         'Doc Ock',  
...         'Green Goblin'  
...       }  
...     )  
...   );  
{default::Movie {id: 9b1cf9e6-3e95-11ec-95a2-138eeb32759c}}
```

To assign to the `Movie.characters` link, we're using a *subquery*. This subquery is executed and resolves to a singleton set of type `Person`, which is assignable to `characters`. Note that the inner `select Person` statement is wrapped in parentheses; this is required for all subqueries in EdgeQL.

Now let's assign to a *single link*.

```
db> insert Villain {
...   name := "Doc Ock",
...   nemesis := (select Hero filter .name = "Spider-Man")
... };
```

This query is valid because the inner subquery is guaranteed to return at most one Hero object, due to the uniqueness constraint on `Hero.name`. If you are filtering on a non-exclusive property, use `assert_single` to guarantee that the subquery will return zero or one results. If more than one result is returned, this query will fail at runtime.

```
db> insert Villain {
...   name := "Doc Ock",
...   nemesis := assert_single((select Hero
...     filter .secret_identity = "Peter B. Parker"
...   ))
... };
```

### 3.7.3 Nested inserts

Just as we used subqueries to populate links with existing objects, we can also execute *nested inserts*.

```
db> insert Villain {
...   name := "The Mandarin",
...   nemesis := (insert Hero {
...     name := "Shang-Chi",
...     secret_identity := "Shaun"
...   })
... };
{default::Villain {id: d47888a0-3e7b-11ec-af13-fb68c8777851}}
```

Now let's write a nested insert for a `multi` link.

```
db> insert Movie {
...   title := "Black Widow",
...   characters := {
...     (select Hero filter .name = "Black Widow"),
...     (insert Hero { name := "Yelena Belova"}),
...     (insert Villain {
...       name := "Dreykov",
...       nemesis := (select Hero filter .name = "Black Widow")
...     })
...   }
... };
```

{default::Movie {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}

We are using *set literal syntax* to construct a set literal containing several `select` and `insert` subqueries. This set contains a mix of `Hero` and `Villain` objects; since these are both subtypes of `Person` (the expected type of `Movie.characters`), this is valid.

You also can't *assign* to a computed property or link; these fields don't actually exist in the database.

```
db> insert Hero {
...   name := "Ant-Man",
```

(continues on next page)

(continued from previous page)

```
...   villains := (select Villain)
... };
error: QueryError: modification of computed link 'villains' of object type
'default::Hero' is prohibited
```

### 3.7.4 With block

In the previous query, we selected Black Widow twice: once in the `characters` set and again as the `nemesis` of Dreykov. In circumstances like this, pulling a subquery into a `with` block lets you avoid duplication.

```
db> with black_widow := (select Hero filter .name = "Black Widow")
...   insert Movie {
...     title := "Black Widow",
...     characters := {
...       black_widow,
...       (insert Hero { name := "Yelena Belova"}),
...       (insert Villain {
...         name := "Dreykov",
...         nemesis := black_widow
...       })
...     }
...   };
{default::Movie {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
```

The `with` block can contain an arbitrary number of clauses; later clauses can reference earlier ones.

```
db> with
...   black_widow := (select Hero filter .name = "Black Widow"),
...   yelena := (insert Hero { name := "Yelena Belova"}),
...   dreykov := (insert Villain {name := "Dreykov", nemesis := black_widow})
...   insert Movie {
...     title := "Black Widow",
...     characters := { black_widow, yelena, dreykov }
...   };
{default::Movie {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
```

### 3.7.5 Conflicts

EdgeDB provides a general-purpose mechanism for gracefully handling possible exclusivity constraint violations. Consider a scenario where we are trying to `insert` Eternals (the `Movie`), but we can't remember if it already exists in the database.

```
db> insert Movie {
...   title := "Eternals"
... }
... unless conflict on .title
... else (select Movie);
{default::Movie {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
```

This query attempts to `insert` Eternals. If it already exists in the database, it will violate the uniqueness constraint on `Movie.title`, causing a *conflict* on the `title` field. The `else` clause is then executed and returned instead. In essence, `unless conflict` lets us “catch” exclusivity conflicts and provide a fallback expression.

---

**Note:** Note that the `else` clause is simply `select Movie`. There’s no need to apply additional filters on `Movie`; in the context of the `else` clause, `Movie` is bound to the conflicting object.

---

**Note:** Using `unless conflict` on *multi properties* is only supported in 2.10 and later.

---

### 3.7.5.1 Upserts

There are no limitations on what the `else` clause can contain; it can be any EdgeQL expression, including an `update` statement. This lets you express *upsert* logic in a single EdgeQL query.

```
db> with
...   title := "Eternals",
...   release_year := 2021
... insert Movie {
...   title := title,
...   release_year := release_year
... }
... unless conflict on .title
... else (
...   update Movie set { release_year := release_year }
... );
{default::Movie {id: f1bf5ac0-3e9d-11ec-b78d-c7dfb363362c}}
```

When a conflict occurs during the initial `insert`, the statement falls back to the `update` statement in the `else` clause. This updates the `release_year` of the conflicting object.

To learn to use upserts by trying them yourself, see [our interactive upserts tutorial](#).

### 3.7.5.2 Suppressing failures

The `else` clause is optional; when omitted, the `insert` statement will return an *empty set* if a conflict occurs. This is a common way to prevent `insert` queries from failing on constraint violations.

```
db> insert Hero { name := "The Wasp" } # initial insert
... unless conflict;
{default::Hero {id: 35b97a92-3e9b-11ec-8e39-6b9695d671ba}}
db> insert Hero { name := "The Wasp" } # The Wasp now exists
... unless conflict;
{}
```

### 3.7.6 Bulk inserts

Bulk inserts are performed by passing in a JSON array as a *query parameter*, *unpacking* it, and using a *for loop* to insert the objects.

```
db> with
...   raw_data := <json>$data,
... for item in json_array_unpack(raw_data) union (
...   insert Hero { name := <str>item['name'] }
... );
Parameter <json>$data: [{"name": "Sersi"}, {"name": "Ikaris"}, {"name": "Thena"}]
{
  default::Hero {id: 35b97a92-3e9b-11ec-8e39-6b9695d671ba},
  default::Hero {id: 35b97a92-3e9b-11ec-8e39-6b9695d671ba},
  default::Hero {id: 35b97a92-3e9b-11ec-8e39-6b9695d671ba},
  ...
}
```

#### See also

<a href="#">Reference &gt; Commands &gt; Insert</a>
<a href="#">Cheatsheets &gt; Inserting data</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Insert</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Upsert</a>

## 3.8 Update

The `update` command is used to update existing objects.

```
db> update Hero
... filter .name = "Hawkeye"
... set { name := "Ronin" };
{default::Hero {id: d476b12e-3e7b-11ec-af13-2717f3dc1d8a}}
```

If you omit the `filter` clause, all objects will be updated. This is useful for updating values across all objects of a given type. The example below cleans up all `Hero.name` values by trimming whitespace and converting them to title case.

```
db> update Hero
... set { name := str_trim(str_title(.name)) };
{default::Hero {id: d476b12e-3e7b-11ec-af13-2717f3dc1d8a}}
```

## 3.8.1 Syntax

The structure of the update statement (`update...filter...set`) is an intentional inversion of SQL's `UPDATE...SET...WHERE` syntax. Curiously, in SQL, the `where` clauses typically occur *last* despite being applied before the `set` statement. EdgeQL is structured to reflect this; first, a target set is specified, then filters are applied, then the data is updated.

### 3.8.1.1 Updating links

When updating links, the `:=` operator will *replace* the set of linked values.

```
db> update movie
... filter .title = "Black Widow"
... set {
...   characters := (
...     select Person
...       filter .name in { "Black Widow", "Yelena", "Dreykov" }
...   )
... };
{default::Title {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
db> select Movie { num_characters := count(.characters) }
... filter .title = "Black Widow";
{default::Movie {num_characters: 3}}
```

To add additional linked items, use the `+=` operator.

```
db> update Movie
... filter .title = "Black Widow"
... set {
...   characters += (insert Villain {name := "Taskmaster"})
... };
{default::Title {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
db> select Movie { num_characters := count(.characters) }
... filter .title = "Black Widow";
{default::Movie {num_characters: 4}}
```

To remove items, use `--`.

```
db> update Movie
... filter .title = "Black Widow"
... set {
...   characters -= Villain # remove all villains
... };
{default::Title {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
db> select Movie { num_characters := count(.characters) }
... filter .title = "Black Widow";
{default::Movie {num_characters: 2}}
```

### 3.8.1.2 With blocks

All top-level EdgeQL statements (`select`, `insert`, `update`, and `delete`) can be prefixed with a `with` block. This is useful for updating the results of a complex query.

```
db> with people := (
...     select Person
...     order by .name
...     offset 3
...     limit 3
...   )
... update people
... set { name := str_trim(.name) };
{
    default::Hero {id: d4764c66-3e7b-11ec-af13-df1ba5b91187},
    default::Hero {id: d7d7e0f6-40ae-11ec-87b1-3f06bed494b9},
    default::Villain {id: d477a836-3e7b-11ec-af13-4fea611d1c31},
}
```

---

**Note:** You can pass any object-type expression into `update`, including polymorphic ones (as above).

---

### 3.8.1.3 See also

For documentation on performing *upsert* operations, see [EdgeQL > Insert > Upserts](#).

<a href="#">Reference &gt; Commands &gt; Update</a>
<a href="#">Cheatsheets &gt; Updating data</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Update</a>

## 3.9 Delete

The `delete` command is used to delete objects from the database.

```
delete Hero
filter .name = 'Iron Man';
```

### 3.9.1 Clauses

Deletion statements support `filter`, `order by`, and `offset/limit` clauses. See [EdgeQL > Select](#) for full documentation on these clauses.

```
delete Hero
filter .name ilike 'the %'
order by .name
offset 10
limit 5;
```

### 3.9.2 Link deletion

Every link is associated with a *link deletion policy*. By default, it isn't possible to delete an object linked to by another.

```
db> delete Hero filter .name = "Yelena Belova";
ConstraintViolationError: deletion of default::Hero
(af7076e0-3e98-11ec-abb3-b3435bbe7c7e) is prohibited by link target policy
{}
```

This deletion failed because Yelena is still in the `characters` list of the Black Widow movie. We must destroy this link before Yelena can be deleted.

```
db> update Movie
... filter .title = "Black Widow"
... set {
...     characters -= (select Hero filter .name = "Yelena Belova")
... };
{default::Movie {id: af706c7c-3e98-11ec-abb3-4bbf3f18a61a}}
db> delete Hero filter .name = "Yelena Belova";
{default::Hero {id: af7076e0-3e98-11ec-abb3-b3435bbe7c7e}}
```

To avoid this behavior, we could update the `Movie.characters` link to use the `allow` deletion policy.

```
type Movie {
    required property title -> str { constraint exclusive };
    required property release_year -> int64;
-   multi link characters -> Person;
+   multi link characters -> Person {
+       on target delete allow;
+   };
}
```

```
type Movie {
    required title: str { constraint exclusive };
    required release_year: int64;
-   multi characters: Person;
+   multi characters: Person {
+       on target delete allow;
+   };
}
```

#### 3.9.2.1 Cascading deletes

If a link uses the `delete source` policy, then deleting a *target* of the link will also delete the object that links to it (*the source*). This behavior can be used to implement cascading deletes; be careful with this power!

The full list of deletion policies is documented at [Schema > Links](#).

### 3.9.3 Return value

A `delete` statement returns the set of deleted objects. You can pass this set into `select` to fetch properties and links of the (now-deleted) objects. This is the last moment this data will be available before being permanently deleted.

```
db> with movie := (delete Movie filter .title = "Untitled")
... select movie {id, title};
{default::Movie {
    id: b11303c6-40ac-11ec-a77d-d393cdedde83,
    title: 'Untitled',
}}
```

See also
<a href="#">Reference &gt; Commands &gt; Delete</a>
<a href="#">Cheatsheets &gt; Deleting data</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Delete</a>

## 3.10 For

EdgeQL supports a top-level `for` statement. These “for loops” iterate over each element of some input set, execute some expression with it, and merge the results into a single output set.

```
db> for number in {0, 1, 2, 3}
... union (
...   select {number, number + 0.5}
... );
{0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5}
```

This statement iterates through each number in the set. Inside the loop, the `number` variable is bound to a singleton set. The inner expression is executed for every element of the input set, and the results of each execution are merged into a single output set.

---

**Note:** The `union` keyword is a required part of the `for` statement syntax; it is intended to indicate explicitly that the results of each loop execution are ultimately merged.

---

### 3.10.1 Bulk inserts

The `for` statement is commonly used for bulk inserts.

```
db> for hero_name in {'Cersi', 'Ikaris', 'Thena'}
... union (
...   insert Hero {name := hero_name}
... );
{
  default::Hero {id: d7d7e0f6-40ae-11ec-87b1-3f06bed494b9},
  default::Hero {id: d7d7f870-40ae-11ec-87b1-f712a4efc3a5},
  default::Hero {id: d7d7f8c0-40ae-11ec-87b1-6b8685d56610}
}
```

This statement iterates through each name in the list of names. Inside the loop, `hero_name` is bound to a `str` singleton, so it can be assigned to `Hero.name`.

Instead of literal sets, it's common to use a `json` parameter for bulk inserts. This value is then “unpacked” into a set of `json` elements and used inside the `for` loop:

```
db> with
...   raw_data := <json>$data,
...   for item in json_array_unpack(raw_data) union (
...     insert Hero { name := <str>item['name'] }
... );
Parameter <json>$data: [{"name": "Sersi"}, {"name": "Ikaris"}, {"name": "Thena"}]
{
  default::Hero {id: d7d7e0f6-40ae-11ec-87b1-3f06bed494b9},
  default::Hero {id: d7d7f870-40ae-11ec-87b1-f712a4efc3a5},
  default::Hero {id: d7d7f8c0-40ae-11ec-87b1-6b8685d56610}
}
```

A similar approach can be used for bulk updates.

See also
<a href="#">Reference &gt; Commands &gt; For</a>

## 3.11 Group

EdgeQL supports a top-level `group` statement. This is used to partition sets into subsets based on some parameters. These subsets then can be additionally aggregated to provide some analytics.

The most basic format is just using the bare `group` to group a set of objects by some property:

```
db> group Movie by .release_year;
{
  {
    key: {release_year: 2016},
    grouping: {'release_year'},
    elements: {
      default::Movie {title: 'Captain America: Civil War'},
      default::Movie {title: 'Doctor Strange'},
    },
  },
  {
    key: {release_year: 2017},
    grouping: {'release_year'},
    elements: {
      default::Movie {title: 'Spider-Man: Homecoming'},
      default::Movie {title: 'Thor: Ragnarok'},
    },
  },
  {
    key: {release_year: 2018},
    grouping: {'release_year'},
    elements: {default::Movie {title: 'Ant-Man and the Wasp'}}},
```

(continues on next page)

(continued from previous page)

```

},
{
  key: {release_year: 2019},
  grouping: {'release_year'},
  elements: {default::Movie {title: 'Spider-Man: No Way Home'}},
},
{
  key: {release_year: 2021},
  grouping: {'release_year'},
  elements: {default::Movie {title: 'Black Widow'}},
},
...
}

```

Notice that the result of `group` is a set of *free objects* with three fields:

- `key`: another free object containing the specific value of the grouping parameter for a given subset.
- `grouping`: set of names of grouping parameters, i.e. the specific names that also appear in the `key` free object.
- `elements`: the actual subset of values that match the `key`.

In the `group` statement, referring to the property in the `by` clause **must** be done by using the leading dot shorthand `.release_year`. The property name then shows up in `grouping` and `key` to indicate the defining characteristics of the particular result. Alternatively, we can give it an alias in an optional `using` clause and then that alias can be used in the `by` clause and will appear in the results:

```

db> group Movie {title}
...  using year := .release_year by year;
{
  {
    key: {year: 2016},
    grouping: {'year'},
    elements: {
      default::Movie {title: 'Captain America: Civil War'},
      default::Movie {title: 'Doctor Strange'},
    },
  },
  {
    key: {year: 2017},
    grouping: {'year'},
    elements: {
      default::Movie {title: 'Spider-Man: Homecoming'},
      default::Movie {title: 'Thor: Ragnarok'},
    },
  },
  {
    key: {year: 2018},
    grouping: {'year'},
    elements: {default::Movie {title: 'Ant-Man and the Wasp'}},
  },
  {
    key: {year: 2019},
    grouping: {'year'},
  }
}

```

(continues on next page)

(continued from previous page)

```

elements: {default::Movie {title: 'Spider-Man: No Way Home'}},
},
{
  key: {year: 2021},
  grouping: {'year'},
  elements: {default::Movie {title: 'Black Widow'}},
},
...
}

```

The `using` clause is perfect for defining a more complex expression to group things by. For example, instead of grouping by the `release_year` we can group by the release decade:

```

db> group Movie {title}
... using decade := .release_year // 10
... by decade;
{
{
  {
    key: {decade: 200},
    grouping: {'decade'},
    elements: {
      default::Movie {title: 'Spider-Man'},
      default::Movie {title: 'Spider-Man 2'},
      default::Movie {title: 'Spider-Man 3'},
      default::Movie {title: 'Iron Man'},
      default::Movie {title: 'The Incredible Hulk'},
    },
  },
  {
    key: {decade: 201},
    grouping: {'decade'},
    elements: {
      default::Movie {title: 'Iron Man 2'},
      default::Movie {title: 'Thor'},
      default::Movie {title: 'Captain America: The First Avenger'},
      default::Movie {title: 'The Avengers'},
      default::Movie {title: 'Iron Man 3'},
      default::Movie {title: 'Thor: The Dark World'},
      default::Movie {title: 'Captain America: The Winter Soldier'},
      default::Movie {title: 'Ant-Man'},
      default::Movie {title: 'Captain America: Civil War'},
      default::Movie {title: 'Doctor Strange'},
      default::Movie {title: 'Spider-Man: Homecoming'},
      default::Movie {title: 'Thor: Ragnarok'},
      default::Movie {title: 'Ant-Man and the Wasp'},
      default::Movie {title: 'Spider-Man: No Way Home'},
    },
  },
  {
    key: {decade: 202},
    grouping: {'decade'},
  }
}

```

(continues on next page)

(continued from previous page)

```
elements: {default::Movie {title: 'Black Widow'}},
},
}
```

It's also possible to group by more than one parameter, so we can group by whether the movie `title` contains a colon *and* the decade it was released. Additionally, let's only consider more recent movies, say, released after 2015, so that we're not overwhelmed by all the combination of results:

```
db> with
...    # Apply the group query only to more recent movies
...    M := (select Movie filter .release_year > 2015)
...    group M {title}
...    using
...        decade := .release_year // 10,
...        has_colon := .title like '%:%'
...    by decade, has_colon;
{
{
    key: {decade: 201, has_colon: false},
    grouping: {'decade', 'has_colon'},
    elements: {
        default::Movie {title: 'Ant-Man and the Wasp'},
        default::Movie {title: 'Doctor Strange'},
    },
},
{
    key: {decade: 201, has_colon: true},
    grouping: {'decade', 'has_colon'},
    elements: {
        default::Movie {title: 'Captain America: Civil War'},
        default::Movie {title: 'Spider-Man: No Way Home'},
        default::Movie {title: 'Thor: Ragnarok'},
        default::Movie {title: 'Spider-Man: Homecoming'},
    },
},
{
    key: {decade: 202, has_colon: false},
    grouping: {'decade', 'has_colon'},
    elements: {default::Movie {title: 'Black Widow'}},
},
}
```

Once we break a set into partitions, we can also use `aggregate` functions to provide some analytics about the data. For example, for the above partitioning (by decade and presence of : in the `title`) we can calculate how many movies are in each subset as well as the average number of words in the movie titles:

```
db> with
...    # Apply the group query only to more recent movies
...    M := (select Movie filter .release_year > 2015),
...    groups := (
...        group M {title}
...        using
```

(continues on next page)

(continued from previous page)

```

...
    decade := .release_year // 10 - 200,
...
    has_colon := .title like '%:%'
...
    by decade, has_colon
...
)
...
select groups {
...
    key := .key {decade, has_colon},
...
    count := count(.elements),
...
    avg_words := math::mean(
        len(str_split(.elements.title, ' '))
    );
{
    {key: {decade: 1, has_colon: false}, count: 2, avg_words: 3},
    {key: {decade: 1, has_colon: true}, count: 4, avg_words: 3},
    {key: {decade: 2, has_colon: false}, count: 1, avg_words: 2},
}

```

**Note:** It is possible to produce results that are grouped in multiple different ways using *grouping sets*. This may be useful in more sophisticated analytics.

<b>See also</b>
<a href="#">Reference &gt; Commands &gt; Group</a>

## 3.12 With

All top-level EdgeQL statements (`select`, `insert`, `update`, and `delete`) can be prefixed by a `with` block. These blocks contain declarations of standalone expressions that can be used in your query.

```

db> with my_str := "hello world"
...
  select str_title(my_str);
{'Hello World'}

```

The `with` clause can contain more than one variable. Earlier variables can be referenced by later ones. Taken together, it becomes possible to write “script-like” queries that execute several statements in sequence.

```

db> with a := 5,
...
  b := 2,
...
  c := a ^ b
...
  select c;
{25}

```

### 3.12.1 Subqueries

There's no limit to the complexity of computed expressions. EdgeQL is fully composable; queries can simply be embedded inside each other. The following query fetches a list of all movies featuring at least one of the original six Avengers.

```
db> with avengers := (select Hero filter .name in {
...     'Iron Man',
...     'Black Widow',
...     'Captain America',
...     'Thor',
...     'Hawkeye',
...     'The Hulk'
...   })
...   select Movie {title}
...   filter avengers in .characters;
{

default::Movie {title: 'Iron Man'},
default::Movie {title: 'The Incredible Hulk'},
default::Movie {title: 'Iron Man 2'},
default::Movie {title: 'Thor'},
default::Movie {title: 'Captain America: The First Avenger'},
...
}
```

### 3.12.2 Query parameters

A common use case for `with` clauses is the initialization of *query parameters*.

```
with user_id := <uuid>$user_id
select User { name }
filter .id = user_id;
```

For a full reference on using query parameters, see [EdgeQL > Parameters](#).

### 3.12.3 Module selection

By default, the *active module* is `default`, so all schema objects inside this module can be referenced by their *short name*, e.g. `User`, `BlogPost`, etc. To reference objects in other modules, we must use fully-qualified names (`default::Hero`).

However, `with` clauses also provide a mechanism for changing the *active module* on a per-query basis.

```
db> with module schema
...   select ObjectType;
```

This `with module` clause changes the default module to `schema`, so we can refer to `schema::ObjectType` (a built-in EdgeDB type) as simply `ObjectType`.

See also
<a href="#">Reference &gt; Commands &gt; With</a>

## 3.13 Path resolution

Element-wise operations with multiple arguments in EdgeDB are generally applied to the *cartesian product* of all the input sets.

```
db> select {'aaa', 'bbb'} ++ {'ccc', 'ddd'};
{'aaaccc', 'aaaddd', 'bbbccc', 'bbbddd'}
```

In some cases, this works out fine, but in others it doesn't make sense. Take this example:

```
select User.first_name ++ ' ' ++ User.last_name;
```

Should the result of this query be every `first_name` in your `User` objects concatenated with every `last_name`? That's probably not what you want.

This is why, in cases where multiple element-wise arguments share a common path (`User.` in this example), EdgeDB factors out the common path rather than using cartesian multiplication.

```
db> select User.first_name ++ ' ' ++ User.last_name;
{'Mina Murray', 'Jonathan Harker', 'Lucy Westenra', 'John Seward'}
```

We assume this is what you want, but if your goal is to get the cartesian product, you can accomplish it one of three ways. You could use `detached`.

```
edgedb> select User.first_name ++ ' ' ++ detached User.last_name;
{
    'Mina Murray',
    'Mina Harker',
    'Mina Westenra',
    'Mina Seward',
    'Jonathan Murray',
    'Jonathan Harker',
    'Jonathan Westenra',
    'Jonathan Seward',
    'Lucy Murray',
    'Lucy Harker',
    'Lucy Westenra',
    'Lucy Seward',
    'John Murray',
    'John Harker',
    'John Westenra',
    'John Seward',
}
```

You could use `with` to attach a different symbol to your set of `User` objects.

```
edgedb> with U := User
..... select U.first_name ++ ' ' ++ User.last_name;
{
    'Mina Murray',
    'Mina Harker',
    'Mina Westenra',
    'Mina Seward',
    'Jonathan Murray',
```

(continues on next page)

(continued from previous page)

```
' Jonathan Harker',
' Jonathan Westenra',
' Jonathan Seward',
' Lucy Murray',
' Lucy Harker',
' Lucy Westenra',
' Lucy Seward',
' John Murray',
' John Harker',
' John Westenra',
' John Seward',
}
```

Or you could leverage the effect scopes have on path resolution. More on that in the *Scopes* section.

The reason `with` works here even though the alias `U` refers to the exact same set is that we only assume you want the path factored in this way when you use the same *symbol* to refer to a set. This means operations with `User.first_name` and `User.last_name` *do* get the common path factored while `U.first_name` and `User.last_name` *do not* and are resolved with cartesian multiplication.

That may leave you still wondering why `U` and `User` did not get a common path factored. `U` is just an alias of `select User` and `User` is the same symbol that we use in our name query. That's true, but EdgeDB doesn't factor in this case because of the queries' scopes.

### 3.13.1 Scopes

Scopes change the way path resolution works. Two sibling select queries — that is, queries at the same level — do not have their paths factored even when they use a common symbol.

```
edgedb> select ((select User.first_name), (select User.last_name));
{
    ('Mina', 'Murray'),
    ('Mina', 'Harker'),
    ('Mina', 'Westenra'),
    ('Mina', 'Seward'),
    ('Jonathan', 'Murray'),
    ('Jonathan', 'Harker'),
    ('Jonathan', 'Westenra'),
    ('Jonathan', 'Seward'),
    ('Lucy', 'Murray'),
    ('Lucy', 'Harker'),
    ('Lucy', 'Westenra'),
    ('Lucy', 'Seward'),
    ('John', 'Murray'),
    ('John', 'Harker'),
    ('John', 'Westenra'),
    ('John', 'Seward'),
}
```

Common symbols in nested scopes *are* factored when they use the same symbol. In this example, the nested queries both use the same `User` symbol as the top-level query. As a result, the `User` in those queries refers to a single object because it has been factored.

```
edgedb> select User {
..... name:= (select User.first_name) ++ ' ' ++ (select User.last_name)
..... };
{
  default::User {name: 'Mina Murray'},
  default::User {name: 'Jonathan Harker'},
  default::User {name: 'Lucy Westenra'},
  default::User {name: 'John Seward'}
}
```

If you have two common scopes and only *one* of them is in a nested scope, the paths are still factored.

```
edgedb> select (Person.name, count(Person.friends));
{('Fran', 3), ('Bam', 2), ('Emma', 3), ('Geoff', 1), ('Tyra', 1)}
```

In this example, `count`, like all aggregate function, creates a nested scope, but this doesn't prevent the paths from being factored as you can see from the results. If the paths were *not* factored, the friend count would be the same for all the result tuples and it would reflect the total number of Person objects that are in *all* friends links rather than the number of Person objects that are in the named Person object's friends link.

If you have two aggregate functions creating *sibling* nested scopes, the paths are *not* factored.

```
edgedb> select (array_agg(distinct Person.name), count(Person.friends));
{(['Fran', 'Bam', 'Emma', 'Geoff'], 3)}
```

This query selects a tuple containing two nested scopes. Here, EdgeDB assumes you want an array of all unique names and a count of the total number of people who are anyone's friend.

### 3.13.1.1 Clauses & Nesting

Most clauses are nested and are subjected to the same rules described above: common symbols are factored and assumed to refer to the same object as the outer query. This is because clauses like `filter` and `order by` need to be applied to each value in the result.

The `limit` clause is not nested in the scope because it needs to be applied globally to the entire result set of your query.

## 3.14 Transactions

EdgeQL supports atomic transactions. The transaction API consists of several commands:

**`start transaction`** Start a transaction, specifying the isolation level, access mode (`read only` vs `read write`), and deferrability.

**`declare savepoint`** Establish a new savepoint within the current transaction. A savepoint is an intermediate point in a transaction flow that provides the ability to partially rollback a transaction.

**`release savepoint`** Destroys a savepoint previously defined in the current transaction.

**`rollback to savepoint`** Rollback to the named savepoint. All changes made after the savepoint are discarded. The savepoint remains valid and can be rolled back to again later, if needed.

**`rollback`** Rollback the entire transaction. All updates made within the transaction are discarded.

**`commit`** Commit the transaction. All changes made by the transaction become visible to others and will persist if a crash occurs.

### 3.14.1 Client libraries

There is rarely a reason to use these commands directly. All EdgeDB client libraries provide dedicated transaction APIs that handle transaction creation under the hood.

#### 3.14.1.1 TypeScript/JS

Using an EdgeQL query string:

```
client.transaction(async tx => {
    await tx.execute(`insert Fish { name := 'Wanda' };`);
});
```

Using the querybuilder:

```
const query = e.insert(e.Fish, {
    name: 'Wanda'
});
client.transaction(async tx => {
    await query.run(tx);
});
```

Full documentation at [Client Libraries > TypeScript/JS](#);

#### 3.14.1.2 Python

```
async for tx in client.transaction():
    async with tx:
        await tx.execute("insert Fish { name := 'Wanda' };")
```

Full documentation at [Client Libraries > Python](#);

#### 3.14.1.3 Golang

```
err := client.Tx(ctx, func(ctx context.Context, tx *Tx) error {
    query := "insert Fish { name := 'Wanda' };"
    if e := tx.Execute(ctx, query); e != nil {
        return e
    }
})
```

Full documentation at [Client Libraries > Go](#).

EdgeQL is a next-generation query language designed to match SQL in power and surpass it in terms of clarity, brevity, and intuitiveness. It's used to query the database, insert/update/delete data, modify/introspect the schema, manage transactions, and more.

## 3.15 Design goals

EdgeQL is a spiritual successor to SQL designed with a few core principles in mind.

**Compatible with modern languages.** A jaw-dropping amount of effort has been spent attempting to [bridge the gap](#) between the *relational* paradigm of SQL and the *object-oriented* nature of modern programming languages. EdgeDB sidesteps this problem by modeling data in an *object-relational* way.

**Strongly typed.** EdgeQL is *inextricably tied* to EdgeDB's rigorous object-oriented type system. The type of all expressions is statically inferred by EdgeDB.

**Designed for programmers.** EdgeQL prioritizes syntax over keywords; It uses { curly braces } to define scopes/structures and the *assignment operator* := to set values. The result is a query language that looks more like code and less like word soup.

**Easy deep querying.** EdgeDB's object-relational nature makes it painless to write deep, performant queries that traverse links, no JOINs required.

**Composable.** Unlike SQL, EdgeQL's syntax is readily composable; queries can be cleanly nested without worrying about Cartesian explosion.

---

**Note:** For a detailed writeup on the design of SQL, see [We Can Do Better Than SQL](#) on the EdgeDB blog.

---

## 3.16 Follow along

The best way to learn EdgeQL is to play with it! Use the [online EdgeQL shell](#) to execute any and all EdgeQL snippets in the following pages. Or follow the [Quickstart](#) to spin up an EdgeDB instance on your computer, then open an [interactive shell](#).



---

## CHAPTER FOUR

---

## GUIDES

These guides contain tutorials introducing EdgeDB to newcomers and how-tos providing more experienced users with examples and advice for tackling some common tasks.

If you are new to EdgeDB check out our [Quickstart](#) guide!

### 4.1 Tutorials

#### 4.1.1 Next.js

**edb-alt-title** Building a simple blog application with EdgeDB and Next.js

We're going to build a simple blog application with [Next.js](#) and EdgeDB. Let's start by scaffolding our app with Next.js's `create-next-app` tool. We'll be using TypeScript for this tutorial.

```
$ npx create-next-app --typescript nextjs-blog
```

This will take a minute to run. The scaffolding tool is creating a simple Next.js app and installing all our dependencies for us. Once it's complete, let's navigate into the directory and start the dev server.

```
$ cd nextjs-blog
$ yarn dev
```

Open [localhost:3000](http://localhost:3000) to see the default Next.js homepage. At this point the app's file structure looks like this:

```
README.md
tsconfig.json
package.json
next.config.js
next-env.d.ts
pages
├── _app.tsx
└── api
    └── hello.ts
    └── index.tsx
public
└── favicon.ico
    └── vercel.svg
styles
└── Home.module.css
    └── globals.css
```

There's a custom App component defined in `pages/_app.tsx` that loads some global CSS, plus the homepage at `pages/index.tsx` and a single API route at `pages/api/hello.ts`. The `styles` and `public` directories contain some other assets.

#### 4.1.1.1 Updating the homepage

Let's start by implementing a simple homepage for our blog application using static data. Replace the contents of `pages/index.tsx` with the following.

```
// pages/index.tsx

import type {NextPage} from 'next';
import Head from 'next/head';
import styles from '../styles/Home.module.css';

type Post = {
  id: string;
  title: string;
  content: string;
};

const HomePage: NextPage = () => {
  const posts: Post[] = [
    {
      id: 'post1',
      title: 'This one weird trick makes using databases fun',
      content: 'Use EdgeDB',
    },
    {
      id: 'post2',
      title: 'How to build a blog with EdgeDB and Next.js',
      content: "Let's start by scaffolding our app with `create-next-app`.",
    },
  ];

  return (
    <div className={styles.container}>
      <Head>
        <title>My Blog</title>
        <meta name="description" content="An awesome blog" />
        <link rel="icon" href="/favicon.ico" />
      </Head>

      <main className={styles.main}>
        <h1 className={styles.title}>Blog</h1>
        <div style={{height: '50px'}}></div>
        {posts.map((post) => {
          return (
            <a href={`/post/${post.id}`} key={post.id}>
              <div className={styles.card}>
                <p>{post.title}</p>
              </div>
            </a>
          )
        })}
      </main>
    </div>
  );
}

export default HomePage;
```

(continues on next page)

(continued from previous page)

```

    );
  })}
  </main>
</div>
);
};

export default HomePage;

```

After saving, Next.js should hot-reload, and the homepage should look something like this.

# Blog

This one weird trick makes  
using databases fun

How to build a blog with  
EdgeDB and Next.js

### 4.1.1.2 Initializing EdgeDB

Now let's spin up a database for the app. First, install the edgedb CLI.

#### Linux or macOS

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.edgedb.com | sh
```

#### Windows Powershell

```
PS> iwr https://ps1.edgedb.com -useb | iex
```

Then check that the CLI is available with the `edgedb --version` command. If you get a `Command not found` error, you may need to open a new terminal window before the `edgedb` command is available.

Once the CLI is installed, initialize a project from the application's root directory. You'll be presented with a series of prompts.

```
$ edgedb project init
No `edgedb.toml` found in `~/nextjs-blog` or above
Do you want to initialize a new project? [Y/n]
> Y
Specify the name of EdgeDB instance to use with this project [default:
nextjs-blog]:
> nextjs-blog
Checking EdgeDB versions...
Specify the version of EdgeDB to use with this project [default: 2.x]:
>
```

Project directory	~/nextjs-blog
Project config	~/nextjs-blog/edgedb.toml
Schema dir (empty)	~/nextjs-blog/dbschema
Installation method	portable package
Start configuration	manual
Version	2.x
Instance name	nextjs-blog

```
Initializing EdgeDB instance...
Applying migrations...
Everything is up to date. Revision initial.
Project initialized.
```

This process has spun up an EdgeDB instance called `nextjs-blog` and “linked” it with your current directory. As long as you’re inside that directory, CLI commands and client libraries will be able to connect to the linked instance automatically, without additional configuration.

To test this, run the `edgedb` command to open a REPL to the linked instance.

```
$ edgedb
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb> select 2 + 2;
{4}
>
```

From inside this REPL, we can execute EdgeQL queries against our database. But there’s not much we can do currently, since our database is schemaless. Let’s change that.

The project initialization process also created a new subdirectory in our project called `dbschema`. This is folder that contains everything pertaining to EdgeDB. Currently it looks like this:

```
dbschema
└── default.esdl
    └── migrations
```

The `default.esdl` file will contain our schema. The `migrations` directory is currently empty, but will contain our migration files. Let’s update the contents of `default.esdl` with the following simple blog schema.

```
# dbschema/default.esdl
```

(continues on next page)

(continued from previous page)

```
module default {
    type BlogPost {
        required property title -> str;
        required property content -> str {
            default := ""
        };
    }
}
```

**Note:** EdgeDB lets you split up your schema into different modules but it's common to keep your entire schema in the `default` module.

Save the file, then let's create our first migration.

```
$ edgedb migration create
did you create object type 'default::BlogPost'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00001.edgeql
```

The `dbschema/migrations` directory now contains a migration file called `00001.edgeql`. Currently though, we haven't applied this migration against our database. Let's do that.

```
$ edgedb migrate
Applied m1fee6oypqprreleos5hmivgfqg6zfkgbrowx7sw5jvnicm73hqdq (00001.edgeql)
```

Our database now has a schema consisting of the `BlogPost` type. We can create some sample data from the REPL. Run the `edgedb` command to re-open the REPL.

```
$ edgedb
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb>
```

Then execute the following `insert` statements.

```
edgedb> insert BlogPost {
.....    title := "This one weird trick makes using databases fun",
.....    content := "Use EdgeDB"
.....};
{default::BlogPost {id: 7f301d02-c780-11ec-8a1a-a34776e884a0}}
edgedb> insert BlogPost {
.....    title := "How to build a blog with EdgeDB and Next.js",
.....    content := "Let's start by scaffolding our app..."
.....};
{default::BlogPost {id: 88c800e6-c780-11ec-8a1a-b3a3020189dd}}
```

### 4.1.1.3 Loading posts with an API route

Now that we have a couple posts in the database, let's load them dynamically with a Next.js API route. To do that, we'll need the `edgedb` client library. Let's install that from NPM:

```
$ npm install edgedb
```

Then create a new file at `pages/api/post.ts` and copy in the following code.

```
// pages/api/post.ts

import type {NextApiRequest, NextApiResponse} from 'next';
import {createClient} from 'edgedb';

export const client = createClient();

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const posts = await client.query(`select BlogPost {
    id,
    title,
    content
};`);
  res.status(200).json(posts);
}
```

This file initializes an EdgeDB client, which manages a pool of connections to the database and provides an API for executing queries. We're using the `.query()` method to fetch all the posts in the database with a simple `select` statement.

If you visit `localhost:3000/api/post` in your browser, you should see a plaintext JSON representation of the blog posts we inserted earlier.

To fetch these from the homepage, we'll use `useState`, `useEffect`, and the built-in `fetch` API. At the top of the `HomePage` component in `pages/index.tsx`, replace the static data and add the missing imports.

```
// pages/index.tsx
+ import {useState, useEffect} from 'react';

type Post = {
  id: string;
  title: string;
  content: string;
};

const HomePage: NextPage = () => {
-  const posts: Post[] = [
-    {
-      id: 'post1',
-      title: 'This one weird trick makes using databases fun',
-      content: 'Use EdgeDB',
-    },
-    {
-
```

(continues on next page)

(continued from previous page)

```

-      id: 'post2',
-      title: 'How to build a blog with EdgeDB and Next.js',
-      content: "Let's start by scaffolding our app...",
-    },
-  ];

+  const [posts, setPosts] = useState<Post[] | null>(null);
+  useEffect(() => {
+    fetch('/api/post')
+      .then((result) => result.json())
+      .then(setPosts);
+  }, []);
+  if (!posts) return <p>Loading...</p>

  return <div>...</div>;
}

```

When you refresh the page, you should briefly see a `Loading...` indicator before the homepage renders the (dynamically loaded!) blog posts.

#### 4.1.1.4 Generating the query builder

Since we're using TypeScript, it makes sense to use EdgeDB's powerful query builder. This provides a schema-aware client API that makes writing strongly typed EdgeQL queries easy and painless. The result type of our queries will be automatically inferred, so we won't need to manually type something like `type Post = { id: string; ... }`.

First, install the generator to your project.

```
$ yarn add --dev @edgedb/generate
```

Then generate the query builder with the following command.

```

$ npx @edgedb/generate edgeql-js
Generating query builder...
Detected tsconfig.json, generating TypeScript files.
To override this, use the --target flag.
Run `npx @edgedb/generate --help` for full options.
Introspecting database schema...
Writing files to ./dbschema/edgeql-js
Generation complete!
Checking the generated query builder into version control
is not recommended. Would you like to update .gitignore to ignore
the query builder directory? The following line will be added:

dbschema/edgeql-js

[y/n] (leave blank for "y")
> y

```

This command introspected the schema of our database and generated some code in the `dbschema/edgeql-js` directory. It also asked us if we wanted to add the generated code to our `.gitignore`; typically it's not good practice to include generated files in version control.

Back in `pages/api/post.ts`, let's update our code to use the query builder instead.

```
// pages/api/post.ts

import type {NextApiRequest, NextApiResponse} from 'next';
import {createClient} from 'edgedb';
+ import e, {$infer} from '../../dbschema/edgeql-js';

export const client = createClient();

+ const selectPosts = e.select(e.BlogPost, () => ({
+   id: true,
+   title: true,
+   content: true,
+ }));

+ export type Posts = $infer<typeof selectPosts>;

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
-  const posts = await client.query(`select BlogPost {
-    id,
-    title,
-    content
-  };`);
+  const posts = await selectPosts.run(client);
  res.status(200).json(posts);
}
```

Instead of writing our query as a plain string, we're now using the query builder to declare our query in a code-first way. As you can see we import the query builder as a single default import `e` from the `dbschema/edgeql-js` directory.

We're also using a utility called `$infer` to extract the inferred type of this query. In VSCode you can hover over `Posts` to see what this type is.

```
4
5  export const client = createClient();
6
7  const selectPosts = e.select(e.BlogPost, () => ({
8    id: true, type Posts = {
9    title: true,   id: string;
10   content: t     content: string;
11 });,           title: string;
12           }[])
13 export type Posts = $infer<typeof selectPosts>;
14
15 export default async function handler(
```

Back in `pages/index.tsx`, let's update our code to use the inferred `Posts` type instead of our manual type declaration.

```
// pages/index.tsx

import type {NextPage} from 'next';
import Head from 'next/head';
import {useEffect, useState} from 'react';
import styles from '../styles/Home.module.css';
+ import {Posts} from "./api/post";

- type Post = {
-   id: string;
-   title: string;
-   content: string;
- };

const Home: NextPage = () => {

+   const [posts, setPosts] = useState<Posts | null>(null);
// ...

}
```

Now, when we update our `selectPosts` query, the type of our dynamically loaded `posts` variable will update automatically—no need to keep our type definitions in sync with our API logic!

#### 4.1.1.5 Rendering blog posts

Our homepage renders a list of links to each of our blog posts, but we haven't implemented the page that actually displays the posts. Let's create a new page at `pages/post/[id].tsx`. This is a [dynamic route](#) that includes an `id` URL parameter. We'll use this parameter to fetch the appropriate post from the database.

Create `pages/post/[id].tsx` and add the following code. We're using `getServerSideProps` to load the blog post data server-side, to avoid loading spinners and ensure the page loads fast.

```
import React from 'react';
import {GetServerSidePropsContext, InferGetServerSidePropsType} from 'next';

import {client} from '../api/post';
import e from 'edgeql-js';

export const getServerSideProps = async (
  context?: GetServerSidePropsContext
) => {
  const post = await client
    .select(e.BlogPost, (post) => ({
      id: true,
      title: true,
      content: true,
      filter_single: e.op(
        post.id,
        '=',
        e.uuid(context!.params!.id as string)
      ),
    }))
}
```

(continues on next page)

(continued from previous page)

```
.run(client);
return {props: {post: post!}};
};

export type GetPost = InferGetServerSidePropsType<typeof getServerSideProps>;

const Post: React.FC<GetPost> = (props) => {
  return (
    <div
      style={{
        margin: 'auto',
        width: '100%',
        maxWidth: '600px',
      }}
    >
      <h1 style={{padding: '50px 0px'}}>{props.post.title}</h1>
      <p style={{color: '#666'}}>{props.post.content}</p>
    </div>
  );
};

export default Post;
```

Inside `getServerSideProps` we're extracting the `id` parameter from `context.params` and using it in our EdgeQL query. The query is a `select` query that fetches the `id`, `title`, and `content` of the post with a matching `id`.

We're using Next's `InferGetServerSidePropsType` utility to extract the inferred type of our query and pass it into `React.FC`. Now, if we update our query, the type of the component props will automatically update too. In fact, this entire application is end-to-end typesafe.

Now, click on one of the blog post links on the homepage. This should bring you to `/post/<uuid>`, which should display something like this:

# How to build a blog with EdgeDB and Next.js

Let's start by scaffolding out app with `create-next-app`.

## 4.1.1.6 Deploying to Vercel

### #1 Deploy EdgeDB

First deploy an EdgeDB instance on your preferred cloud provider:

- [AWS](#)
- [Google Cloud](#)
- [Azure](#)
- [DigitalOcean](#)
- [Fly.io](#)
- [Docker](#) (cloud-agnostic)

### #2. Find your instance's DSN

The DSN is also known as a connection string. It will have the format `edgedb://username:password@hostname:port`. The exact instructions for this depend on which cloud you are deploying to.

### #3 Apply migrations

Use the DSN to apply migrations against your remote instance.

```
$ edgedb migrate --dsn <your-instance-dsn> --tls-security insecure
```

---

**Note:** You have to disable TLS checks with `--tls-security insecure`. All EdgeDB instances use TLS by default, but configuring it is out of scope of this project.

---

Once you've applied the migrations, consider creating some sample data in your database. Open a REPL and insert some blog posts:

```
$ edgedb --dsn <your-instance-dsn> --tls-security insecure
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb> insert BlogPost { title := "Test post" };
{default::BlogPost {id: c00f2c9a-cbf5-11ec-8ecb-4f8e702e5789}}
```

#### #4 Set up a `prebuild` script

Add the following `prebuild` script to your `package.json`. When Vercel initializes the build, it will trigger this script which will generate the query builder. The `npx @edgedb/generate edgeql-js` command will read the value of the `EDGEDB_DSN` variable, connect to the database, and generate the query builder before Vercel starts building the project.

```
// package.json
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "lint": "next lint",
+ "prebuild": "npx @edgedb/generate edgeql-js"
},
```

#### #5 Deploy to Vercel

Deploy this app to Vercel with the button below.

When prompted:

- Set `EDGEDB_DSN` to your database's DSN
- Set `EDGEDB_CLIENT_TLS_SECURITY` to `insecure`. This will disable EdgeDB's default TLS checks; configuring TLS is beyond the scope of this tutorial.

## Configure Project

Please provide values for the required Environment Variables.

▼ Required Environment Variables	
NAME	VALUE (WILL BE ENCRYPTED)
EDGEDB_DSN	<code>edgedb://edgedb:edgedbpassword@...</code>
EDGEDB_CLIENT_TLS_SECURITY	insecure

[Deploy](#)

### #6 View the application

Once deployment has completed, view the application at the deployment URL supplied by Vercel.

#### 4.1.1.7 Wrapping up

Admittedly this isn't the prettiest blog of all time, or the most feature-complete. But this tutorial demonstrates how to work with EdgeDB in a Next.js app, including data fetching with API routes and `getServerSideProps`.

The next step is to add a `/newpost` page with a form for writing new blog posts and saving them into EdgeDB. That's left as an exercise for the reader.

To see the final code for this tutorial, refer to [github.com/edgedb/edgedb-examples/tree/main/nextjs-blog](https://github.com/edgedb/edgedb-examples/tree/main/nextjs-blog).

### 4.1.2 FastAPI

#### **edb-alt-title** Building a REST API with EdgeDB and FastAPI

Because FastAPI encourages and facilitates strong typing, it's a natural pairing with EdgeDB. Our Python code generation generates not only typed query functions but result types you can use to annotate your endpoint handler functions.

EdgeDB can help you quickly build REST APIs in Python without getting into the rigmarole of using ORM libraries to handle your data effectively. Here, we'll be using [FastAPI](#) to expose the API endpoints and EdgeDB to store the content.

We'll build a simple event management system where you'll be able to fetch, create, update, and delete `events` and `event hosts` via RESTful API endpoints.

Watch our video tour of this example project to get a preview of what you'll be building in this guide:

#### 4.1.2.1 Prerequisites

Before we start, make sure you've [installed](#) the `edgedb` command line tool. In this tutorial, we'll use Python 3.10 and take advantage of the asynchronous I/O paradigm to communicate with the database more efficiently. If you want to skip ahead, the completed source code for this API can be found [in our examples repo](#).

##### 4.1.2.1.1 Create a project directory

To get started, create a directory for your project and change into it.

```
$ mkdir fastapi-crud  
$ cd fastapi-crud
```

##### 4.1.2.1.2 Install the dependencies

Create a Python 3.10 virtual environment, activate it, and install the dependencies with this command (in Linux/macOS; see the following note for help with Windows):

```
$ python -m venv myvenv  
$ source myvenv/bin/activate  
$ pip install edgedb fastapi 'httpx[cli]' uvicorn
```

---

**Note:** Make sure you run `source myvenv/bin/activate` any time you want to come back to this project to activate its virtual environment. If not, you may start working under your system's default Python environment which could be the incorrect version or not have the dependencies installed. If you want to confirm you're using the right environment, run `which python`. You should see that the current python is inside your venv directory.

---

---

**Note:** The commands will differ for Windows/Powershell users; this guide provides instructions for working with virtual environments across a range of OSes, including Windows.

---

##### 4.1.2.1.3 Initialize the database

Now, let's initialize an EdgeDB project. From the project's root directory:

```
$ edgedb project init  
No `edgedb.toml` found in `<project-path>` or above  
Do you want to initialize a new project? [Y/n]  
> Y  
Specify the name of EdgeDB instance to use with this project [default:  
fastapi_crud]:  
> fastapi_crud  
Checking EdgeDB versions...  
Specify the version of EdgeDB to use with this project [default: 2.7]:  
> 2.7
```

Once you've answered the prompts, a new EdgeDB instance called `fastapi_crud` will be created and started. If you see `Project initialized`, you're ready.

#### 4.1.2.1.4 Connect to the database

Let's test that we can connect to the newly started instance. To do so, run:

```
$ edgedb
```

You should see this prompt indicating you are now connected to your new database instance:

```
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb>
```

You can start writing queries here. Since this database is empty, that won't get you very far, so let's start designing our data model instead.

#### 4.1.2.2 Schema design

The event management system will have two entities: **events** and **users**. Each *event* can have an optional link to a *user* who is that event's host. The goal is to create API endpoints that'll allow us to fetch, create, update, and delete the entities while maintaining their relationships.

EdgeDB allows us to declaratively define the structure of the entities. If you've worked with SQLAlchemy or Django ORM, you might refer to these declarative schema definitions as *models*. In EdgeDB we call them "object types".

The schema lives inside `.esdl` files in the `dbschema` directory. It's common to declare the entire schema in a single file `dbschema/default.esdl`. This file is created for you when you run `edgedb project init`, but you'll need to fill it with your schema. This is what our datatypes look like:

```
# dbschema/default.esdl

module default {
    abstract type Auditable {
        required property created_at -> datetime {
            readonly := true;
            default := datetime_current();
        }
    }

    type User extending Auditable {
        required property name -> str {
            constraint exclusive;
            constraint max_len_value(50);
        };
    }

    type Event extending Auditable {
        required property name -> str {
            constraint exclusive;
            constraint max_len_value(50);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

property address -> str;
property schedule -> datetime;
link host -> User;
}
}

```

Here, we've defined an abstract type called `Auditable` to take advantage of EdgeDB's schema mixin system. This allows us to add a `created_at` property to multiple types without repeating ourselves. Abstract types don't have any concrete footprints in the database, as they don't hold any actual data. Their only job is to propagate properties, links, and constraints to the types that extend them.

The `User` type extends `Auditable` and inherits the `created_at` property as a result. Since `created_at` has a default value, it's auto-filled with the return value of the `datetime_current` function. Along with the property conveyed to it by the extended type, the `User` type defines its own concrete required property called `name`. We impose two constraints on this property: names should be unique (`constraint exclusive`) and shorter than 50 characters (`constraint max_len_value(50)`).

We also define an `Event` type that extends the `Auditable` abstract type. It contains its own concrete properties and links: `address`, `schedule`, and an optional link called `host` that corresponds to a `User`.

#### 4.1.2.3 Run a migration

With the schema created, it's time to lock it in. The first step is to create a migration.

```
$ edgedb migration create
```

When this step is successful, you'll see `Created dbschema/migrations/00001.edgeql`.

Now run the migration we just created.

```
$ edgedb migrate
```

Once this is done, you'll see `Applied` along with the migration's ID. I like to go one step further in verifying success and see the schema applied to my database. To do that, first fire up the EdgeDB console:

```
$ edgedb
```

In the console, type `\ds` (for "describe schema"). If everything worked, we should output very close to the schema we added in the `default.esdl` file:

```

module default {
    abstract type Auditable {
        property created_at -> std::datetime {
            default := (std::datetime_current());
            readonly := true;
        };
    };
    type Event extending default::Auditable {
        link host -> default::User;
        property address -> std::str;
        required property name -> std::str {
            constraint std::exclusive;
            constraint std::max_len_value(50);
        };
    };
}

```

(continues on next page)

(continued from previous page)

```

    };
    property schedule -> std::datetime;
};

type User extending default::Auditable {
    required property name -> std::str {
        constraint std::exclusive;
        constraint std::max_len_value(50);
    };
};

};

```

#### 4.1.2.4 Build the API endpoints

With the schema established, we're ready to start building out the app. Let's start by creating an `app` directory inside our project:

```
$ mkdir app
```

Within this `app` directory, we're going to create three modules: `events.py` and `users.py` which represent the events and users APIs respectively, and `main.py` that registers all the endpoints and exposes them to the `uvicorn` webserver. We also need an `__init__.py` to mark this directory as a package so we can easily import from it. Go ahead and create that file now in your editor or via the command line like this (from the project root):

```
$ touch app/__init__.py
```

We'll work on the users API first since it's the simpler of the two.

##### 4.1.2.4.1 Users API

We want this app to be type safe, end to end. To achieve this, instead of hard-coding string queries into the app, we'll use code generation to generate typesafe functions from queries we write in `.edgeql` files. These files are simple text files containing the queries we want our app to be able to run.

The code generator will search through our project for all files with the `.edgeql` extension and generate those functions for us as individual Python modules. When you installed the EdgeDB client (via `pip install edgedb`), the code generator was installed alongside it, so you're already ready to go. We just need to write those queries!

We'll write queries for one endpoint at a time to start so you can see how the pieces fit together. To keep things organized, create a new directory inside `app` called `queries`. Create a new file in `app/queries` named `get_users.edgeql` and open it in your editor. Write the query into this file. It's the same one we would have written inline in our Python code as shown in the code block above:

```
select User {name, created_at};
```

We need one more query to finish off this endpoint. Create another file inside `app/queries` named `get_user_by_name.edgeql` and open it in your editor. Add this query:

```
select User {name, created_at}
filter User.name = <str>$name
```

Save that file and get ready to kick off the magic that is code generation!

```
$ edgedb-py
Found EdgeDB project: <project-path>
Processing <project-path>/app/queries/get_user_by_name.edgeql
Processing <project-path>/app/queries/get_users.edgeql
Generating <project-path>/app/queries/get_user_by_name.py
Generating <project-path>/app/queries/get_users.py
```

The code generator creates one module per query file by default and places them at the same path as the query files.

With code generated, we're ready to write an endpoint. Let's create the GET `/users` endpoint so that we can request the `User` objects saved in the database. Create a new file `app/users.py`, open it in your editor, and add the following code:

```
# app/users.py
from __future__ import annotations

import datetime
from http import HTTPStatus
from typing import List

import edgedb
from fastapi import APIRouter, HTTPException, Query
from pydantic import BaseModel

from .queries import get_user_by_name_async_edgeql as get_user_by_name_qry
from .queries import get_users_async_edgeql as get_users_qry

router = APIRouter()
client = edgedb.create_async_client()

class RequestData(BaseModel):
    name: str

@router.get("/users")
async def get_users(
    name: str = Query(None, max_length=50)
) -> List[get_users_qry.GetUsersResult] | get_user_by_name_qry.GetUserByNameResult:

    if not name:
        users = await get_users_qry.get_users(client)
        return users
    else:
        user = await get_user_by_name_qry.get_user_by_name(client, name=name)
        return user
```

We've imported the generated code and aliased it (using `as <new-name>`) to make the module names we use in our code a bit neater.

The `APIRouter` instance does the actual work of exposing the API. We also create an `async` EdgeDB client instance to communicate with the database.

By default, this API will return a list of all users, but you can also filter the user objects by name. We have the `requestData` class to handle the data an API consumer will need to send in case they want to get only a single user.

The types we’re using in the return annotation have been generated by the EdgeDB code generation based on the queries we wrote and our database’s schema.

Note that we’re also calling the appropriate generated function based on whether or not the API consumer passes an argument for `name`.

This nearly gets us there but not quite. We have one potential outcome not accounted for: a query for a user by name that returns no results. In that case, we’ll want to return a 404 (not found).

To fix it, we’ll check in the `else` case whether we got anything back from the single user query. If not, we’ll go ahead and raise an exception. This will send the 404 (not found) response to the user.

```
# app/users.py
...
if not name:
    users = await get_users_qry.get_users(client)
    return users
else:
    user = await get_user_by_name_qry.get_user_by_name(client, name=name)
    if not user:
        raise HTTPException(
            status_code=HTTPStatus.NOT_FOUND,
            detail={"error": f"Username '{name}' does not exist."},
        )
    return user
...
```

To summarize, in the `get_users` function, we use our generated code to perform asynchronous queries via the `edgedb` client. Then we return the query results. Afterward, the JSON serialization part is taken care of by FastAPI.

Before we can use this endpoint, we need to expose it to the server. We’ll do that in the `main.py` module. Create `app/main.py` and open it in your editor. Here’s the content of the module:

```
# app/main.py
from __future__ import annotations

from fastapi import FastAPI
from starlette.middleware.cors import CORSMiddleware

from app import users

fast_api = FastAPI()

# Set all CORS enabled origins.
fast_api.add_middleware(
    CORSMiddleware,
    allow_origins=['*'],
    allow_credentials=True,
    allow_methods=['*'],
    allow_headers=['*'],
)

fast_api.include_router(users.router)
```

Here, we import everything we need, including our own `users` module containing the router and endpoint logic for

the users API. We instantiate the API, give it a permissive CORS configuration, and give it the users router.

To test the endpoint, go to the project root and run:

```
$ uvicorn app.main:fast_api --port 5001 --reload
```

This will start a `uvicorn` server and you'll be able to start making requests against it. Earlier, we installed the `HTTPx` client library to make HTTP requests programmatically. It also comes with a neat command-line tool that we'll use to test our API.

While the `uvicorn` server is running, bring up a new console. Activate your virtual environment by running `source myenv/bin/activate` and run:

```
$ httpx -m GET http://localhost:5001/users
```

You'll see the following output on the console:

```
HTTP/1.1 200 OK
date: Sat, 16 Apr 2022 22:58:11 GMT
server: uvicorn
content-length: 2
content-type: application/json

[]
```

---

**Note:** If you find yourself with a result you don't expect when making a request to your API, switch over to the `uvicorn` server console. You should find a traceback that will point you to the problem area in your code.

---

If you see this result, that means the API is working! It's not especially useful though. Our request yields an empty list because the database is currently empty. Let's create the POST `/users` endpoint in `app/users.py` to start saving users in the database. Before we do that though, let's go ahead and create the new query we'll need.

Create and open `app/queries/create_user.edgeql` and fill it with this query:

```
select (insert User {
    name := <str>$name
}) {
    name,
    created_at
};
```

---

**Note:** We're running our `insert` inside a `select` here so that we can return the `name` and `created_at` properties. If we just ran the `insert` bare, it would return only the `id`.

---

Save the file and run `edgedb-py` to generate the new function. Now, we're ready to open `app/users.py` again and add the POST endpoint. First, import the generated function for the new query:

```
# app/users.py
...
from .queries import create_user_async_edgeql as create_user_qry
from .queries import get_user_by_name_async_edgeql as get_user_by_name_qry
from .queries import get_users_async_edgeql as get_users_qry
...
```

Then write the endpoint to call that function:

```
# app/users.py
...
@router.post("/users", status_code=HTTPStatus.CREATED)
async def post_user(user: RequestData) -> create_user_qry.CreateUserResult:

    try:
        created_user = await create_user_qry.create_user(client, name=user.name)
    except edgedb.errors.ConstraintViolationError:
        raise HTTPException(
            status_code=HTTPStatus.BAD_REQUEST,
            detail={"error": f"Username '{user.name}' already exists."},
        )
    return created_user
```

In the above snippet, we ingest data with the shape dictated by the `requestData` model and return a payload of the query results. The `try...except` block gracefully handles the situation where the API consumer might try to create multiple users with the same name. A successful request will yield the status code HTTP 201 (created) along with the new user's `id`, `name`, and `created_at` as JSON.

To test it out, make a request as follows:

```
$ httpx -m POST http://localhost:5001/users \
--json '{"name" : "Jonathan Harker"}'
```

The output should look similar to this:

```
HTTP/1.1 201 Created
...
{
    "id": "53771f56-6f57-11ed-8729-572f5fba7ddc",
    "name": "Jonathan Harker",
    "created_at": "2022-04-16T23:09:30.929664+00:00"
}
```

---

**Note:** Since IDs are generated, your `id` values probably won't match the values in this guide. This is not a problem.

If you try to make the same request again, it'll throw an HTTP 400 (bad request) error:

```
HTTP/1.1 400 Bad Request
...
{
    "detail": {
        "error": "Username 'Jonathan Harker' already exists."
    }
}
```

Before we move on to the next step, create 2 more users called `Count Dracula` and `Mina Murray`. Once you've done that, we can move on to the next step of building the `PUT /users` endpoint to update existing user data.

We'll start again with the query. Create a new file in `app/queries` named `update_user.edgql`. Open it in your editor and enter this query:

```
select (
    update User filter .name = <str>$current_name
        set {name := <str>$new_name}
) {name, created_at};
```

Save the file and generate again using edgedb-py. Now, we'll import that and add the endpoint over in `app/users.py`.

```
# app/users.py
...
from .queries import create_user_async_edgeql as create_user_qry
from .queries import get_user_by_name_async_edgeql as get_user_by_name_qry
from .queries import get_users_async_edgeql as get_users_qry
from .queries import update_user_async_edgeql as update_user_qry
...
@router.put("/users")
async def put_user(
    user: RequestData, current_name: str
) -> UpdateUserResult:
    try:
        updated_user = await update_user_qry.update_user(
            client,
            new_name=user.name,
            current_name=current_name,
        )
    except edgedb.errors.ConstraintViolationError:
        raise HTTPException(
            status_code=HTTPStatus.BAD_REQUEST,
            detail={"error": f"Username '{user.name}' already exists."},
        )
    if not updated_user:
        raise HTTPException(
            status_code=HTTPStatus.NOT_FOUND,
            detail={"error": f"User '{current_name}' was not found."},
        )
    return updated_user
```

Not much new happening here. We wrote our query with a `current_name` parameter for finding the user to be updated. The `user` argument will give us the changes to make to that user, which in this case can only be the `name` since that's the only property a user has. We pull the name out of `user` and pass it as our `new_name` argument to the generated function. The endpoint calls the generated function passing the client and those two values, and the user is updated.

We've accounted for the possibility of a user trying to change a user's name to a new name that conflicts with a different user. That will return a 400 (bad request) error. We've also accounted for the possibility of a user trying to update a user that doesn't exist, which will return a 404 (not found).

Let's save everything and test this out.

```
$ httpx -m PUT http://localhost:5001/users \
    -p 'current_name' 'Jonathan Harker' \
    --json '{"name" : "Dr. Van Helsing"}'
```

This will return:

```
HTTP/1.1 200 OK
...
[{"id": "53771f56-6f57-11ed-8729-572f5fba7ddc",
 "name": "Dr. Van Helsing",
 "created_at": "2022-04-16T23:09:30.929664+00:00"}]
```

If you try to change the name of a user to match that of an existing user, the endpoint will throw an HTTP 400 (bad request) error:

```
$ httpx -m PUT http://localhost:5001/users \
-p 'current_name' 'Count Dracula' \
--json '{"name" : "Dr. Van Helsing"}'
```

This returns:

```
HTTP/1.1 400 Bad Request
...
{
  "detail": {
    "error": "Username 'Dr. Van Helsing' already exists."
  }
}
```

Since we've verified that endpoint is working, let's move on to the `DELETE /users` endpoint. It'll allow us to query the name of the targeted object to delete it.

Start by creating `app/queries/delete_user.edgeql` and filling it with this query:

```
select (
  delete User filter .name = <str>$name
) {name, created_at};
```

Generate the new function by again running `edgeql-py`. Then re-open `app/users.py`. This endpoint's code will look similar to the endpoints we've already written:

```
# app/users.py
...
from .queries import create_user_async_edgeql as create_user_qry
from .queries import delete_user_async_edgeql as delete_user_qry
from .queries import get_user_by_name_async_edgeql as get_user_by_name_qry
from .queries import get_users_async_edgeql as get_users_qry
from .queries import update_user_async_edgeql as update_user_qry
...
@router.delete("/users")
async def delete_user(name: str) -> delete_user_qry.DeleteUserResult:
    try:
        deleted_user = await delete_user_qry.delete_user(
            client,
            name=name,
        )
```

(continues on next page)

(continued from previous page)

```

except edgedb.errors.ConstraintViolationError:
    raise HTTPException(
        status_code=HTTPStatus.BAD_REQUEST,
        detail={"error": "User attached to an event. Cannot delete."},
    )

if not deleted_user:
    raise HTTPException(
        status_code=HTTPStatus.NOT_FOUND,
        detail={"error": f"User '{name}' was not found."},
    )
return deleted_user

```

This endpoint will simply delete the requested user if the user isn't attached to any event. If the targeted object *is* attached to an event, the API will throw an HTTP 400 (bad request) error and refuse to delete the object. To test it out by deleting Count Dracula, on your console, run:

```
$ httpx -m DELETE http://localhost:5001/users \
-p 'name' 'Count Dracula'
```

If it worked, you should see this result:

```
HTTP/1.1 200 OK
...
[
{
    "id": "e6837562-6f55-11ed-8744-ff1b295ed864",
    "name": "Count Dracula",
    "created_at": "2022-04-16T23:23:56.630101+00:00"
}
]
```

With that, you've written the entire users API! Now, we move onto the events API which is slightly more complex. (Nothing you can't handle though. )

#### 4.1.2.4.2 Events API

Let's start with the POST /events endpoint, and then we'll fetch the objects created via POST using the GET /events endpoint.

First, we need a query. Create a file app/queries/create\_event.edgeql and drop this query into it:

```

with name := <str>$name,
      address := <str>$address,
      schedule := <str>$schedule,
      host_name := <str>$host_name

select (
    insert Event {
        name := name,
        address := address,
        schedule := <datetime>schedule,

```

(continues on next page)

(continued from previous page)

```

host := assert_single(
    (select detached User filter .name = host_name)
)
}
) {name, address, schedule, host: {name}};

```

Run edgedb-py to generate a function from that query.

Create a file in app named events.py and open it in your editor. It's time to code up the endpoint to use that freshly generated query.

```

# app/events.py
from __future__ import annotations

from http import HTTPStatus
from typing import List

import edgedb
from fastapi import APIRouter, HTTPException, Query
from pydantic import BaseModel

from .queries import create_event_async_edgeql as create_event_qry

router = APIRouter()
client = edgedb.create_async_client()

class RequestData(BaseModel):
    name: str
    address: str
    schedule: str
    host_name: str

@router.post("/events", status_code=HTTPStatus.CREATED)
async def post_event(event: RequestData) -> create_event_qry.CreateEventResult:
    try:
        created_event = await create_event_qry.create_event(
            client,
            name=event.name,
            address=event.address,
            schedule=event.schedule,
            host_name=event.host_name,
        )

    except edgedb.errors.InvalidValueError:
        raise HTTPException(
            status_code=HTTPStatus.BAD_REQUEST,
            detail={
                "error": "Invalid datetime format."
                "Datetime string must look like this: "
                "'2010-12-27T23:59:59-07:00'",
            },

```

(continues on next page)

(continued from previous page)

```

)
except edgedb.errors.ConstraintViolationError:
    raise HTTPException(
        status_code=HTTPStatus.BAD_REQUEST,
        detail=f"Event name '{event.name}' already exists",
)
return created_event

```

Like the POST `/users` endpoint, the incoming and outgoing shape of the POST `/events` endpoint's data are defined by the `requestData` model and the generated `CreateEventResult` model respectively. The `post_events` function asynchronously inserts the data into the database and returns the fields defined in the `select` query we wrote earlier, along with the new event's `id`.

The exception handling logic validates the shape of the incoming data. For example, just as before in the users API, the events API will complain if you try to create multiple events with the same name. Also, the field `schedule` accepts data as an ISO 8601 timestamp string. Values not adhering to that will incur an HTTP 400 (bad request) error.

It's almost time to test, but before we can do that, we need to expose this new API in `app/main.py`. Open that file, and update the import on line 6 to also import `events`:

```
# app/main.py
...
from app import users, events
...
```

Drop down to the bottom of `main.py` and include the events router:

```
# app/main.py
...
fast_api.include_router(events.router)
```

Let's try it out. Here's how you'd create an event:

```
$ httpx -m POST http://localhost:5001/events \
--json '{
    "name": "Resuscitation",
    "address": "Britain",
    "schedule": "1889-07-27T23:59:59-07:00",
    "host_name": "Mina Murray"
}'
```

If everything worked, you'll see output like this:

```
HTTP/1.1 200 OK
...
{
    "id": "0b1847f4-6f3d-11ed-9f27-6fcdf20ffe22",
    "name": "Resuscitation",
    "address": "Britain",
    "schedule": "1889-07-28T06:59:59+00:00",
    "host": {
        "name": "Mina Murray"
```

(continues on next page)

(continued from previous page)

```

    }
}
```

To speed this up a bit, we'll go ahead and write all the remaining queries in one shot. Then we can flip back to `app/events.py` and code up all the endpoints. Start by creating a file in `app/queries` named `get_events.edgeql`. This one is really straightforward:

```
select Event {name, address, schedule, host : {name}};
```

Save that one and create `app/queries/get_event_by_name.edgeql` with this query:

```
select Event {
    name, address, schedule,
    host : {name}
} filter .name = <str>$name;
```

Those two will handle queries for `GET /events`. Next, create `app/queries/update_event.edgeql` with this query:

```
with current_name := <str>$current_name,
      new_name := <str>$name,
      address := <str>$address,
      schedule := <str>$schedule,
      host_name := <str>$host_name

select (
    update Event filter .name = current_name
    set {
        name := new_name,
        address := address,
        schedule := <datetime>schedule,
        host := (select User filter .name = host_name)
    }
) {name, address, schedule, host: {name}};
```

That query will handle `PUT` requests. The last method left is `DELETE`. Create `app/queries/delete_event.edgeql` and put this query in it:

```
select (
    delete Event filter .name = <str>$name
) {name, address, schedule, host : {name}};
```

Run `edgedb-py` to generate the new functions. Open `app/events.py` so we can start getting these functions implemented in the API! We'll start by coding `GET`. Import the newly generated queries and write the `GET` endpoint in `events.py`:

```
# app/events.py
...
from .queries import create_event_async_edgeql as create_event_qry
from .queries import delete_event_async_edgeql as delete_event_qry
from .queries import get_event_by_name_async_edgeql as get_event_by_name_qry
from .queries import get_events_async_edgeql as get_events_qry
from .queries import update_event_async_edgeql as update_event_qry
...
```

(continues on next page)

(continued from previous page)

```
@router.get("/events")
async def get_events(
    name: str = Query(None, max_length=50)
) -> List[get_events_qry.GetEventsResult] | get_event_by_name_qry.GetEventByNameResult:
    if not name:
        events = await get_events_qry.get_events(client)
        return events
    else:
        event = await get_event_by_name_qry.get_event_by_name(client, name=name)
        if not event:
            raise HTTPException(
                status_code=HTTPStatus.NOT_FOUND,
                detail={"error": f"Event '{name}' does not exist."},
            )
    return event
```

Save that file and test it like this:

```
$ httpx -m GET http://localhost:5001/events
```

We should get back an array containing all our events (which, at the moment, is just the one):

```
HTTP/1.1 200 OK
...
[
    {
        "id": "0b1847f4-6f3d-11ed-9f27-6fcdf20ffe22",
        "name": "Resuscitation",
        "address": "Britain",
        "schedule": "1889-07-28T06:59:59+00:00",
        "host": {
            "name": "Mina Murray"
        }
    }
]
```

You can also use the GET `/events` endpoint to return a single event object by name. To locate the `Resuscitation` event, you'd use the `name` parameter with the GET API as follows:

```
$ httpx -m GET http://localhost:5001/events \
    -p 'name' 'Resuscitation'
```

That'll return a result that looks like the response we just got without the `name` parameter, except that it's a single object instead of an array.

```
HTTP/1.1 200 OK
...
{
    "id": "0b1847f4-6f3d-11ed-9f27-6fcdf20ffe22",
    "name": "Resuscitation",
    "address": "Britain",
    "schedule": "1889-07-28T06:59:59+00:00",
    "host": {
```

(continues on next page)

(continued from previous page)

```

    "name": "Mina Murray"
}
}
```

If we'd had multiple events, the response to our first test would have given us all of them.

Let's finish off the events API with the PUT and DELETE endpoints. Open `app/events.py` and add this code:

```

# app/events.py
...
@router.put("/events")
async def put_event(
    event: RequestData, current_name: str
) -> update_event_qry.UpdateEventResult:
    try:
        updated_event = await update_event_qry.update_event(
            client,
            current_name=current_name,
            name=event.name,
            address=event.address,
            schedule=event.schedule,
            host_name=event.host_name,
        )

    except edgedb.errors.InvalidValueError:
        raise HTTPException(
            status_code=HTTPStatus.BAD_REQUEST,
            detail={
                "error": "Invalid datetime format. "
                "Datetime string must look like this: '2010-12-27T23:59:59-07:00'",
            },
        )

    except edgedb.errors.ConstraintViolationError:
        raise HTTPException(
            status_code=HTTPStatus.BAD_REQUEST,
            detail={"error": f"Event name '{event.name}' already exists."},
        )

    if not updated_event:
        raise HTTPException(
            status_code=HTTPStatus.INTERNAL_SERVER_ERROR,
            detail={"error": f"Update event '{event.name}' failed."},
        )

    return updated_event

@router.delete("/events")
async def delete_event(name: str) -> delete_event_qry.DeleteEventResult:
    deleted_event = await delete_event_qry.delete_event(client, name=name)

    if not deleted_event:

```

(continues on next page)

(continued from previous page)

```

raise HTTPException(
    status_code=HTTPStatus.INTERNAL_SERVER_ERROR,
    detail={"error": f"Delete event '{name}' failed."},
)

return deleted_event

```

The events API is now ready to handle updates and deletion. Let's try out a cool alternative way to test these new endpoints.

#### 4.1.2.4.3 Browse the endpoints using the native OpenAPI doc

FastAPI automatically generates OpenAPI schema from the API endpoints and uses those to build the API docs. While the uvicorn server is running, go to your browser and head over to <http://localhost:5001/docs>. You should see an API navigator like this:



### default

<b>GET</b>	/events	Get Events	▼
<b>PUT</b>	/events	Put Event	▼
<b>POST</b>	/events	Post Event	▼
<b>DELETE</b>	/events	Delete Event	▼
<b>GET</b>	/users	Get Users	▼
<b>PUT</b>	/users	Put User	▼
<b>POST</b>	/users	Post User	▼
<b>DELETE</b>	/users	Delete User	▼

This documentation allows you to play with the APIs interactively. Let's try to make a request to the PUT /events. Click on the API that you want to try and then click on the **Try it out** button. You can do it in the UI as follows:

The screenshot shows the EdgeDB API documentation interface. At the top, it says "PUT /events Put Event". Below that, there's a "Parameters" section with one entry: "current\_name \* required string (query)" with the value "Resuscitation". There are "Cancel" and "Reset" buttons. Under "Request body required", it says "application/json" and shows a JSON object:

```
{
  "name": "Dr. Seward's Birthday",
  "address": "Britain",
  "schedule": "1889-07-28T06:59:59+00:00",
  "host_name": "Mina Murray"
}
```

Clicking the **execute** button will make the request and return the following payload:

The screenshot shows the response for a successful PUT request. It includes a "Code" column with "200" and a "Details" column. The "Response body" section shows the JSON payload from the previous screenshot. The "Response headers" section shows the following HTTP headers:

```
access-control-allow-credentials: true
access-control-allow-origin: *
content-length: 209
content-type: application/json
date: Mon, 28 Nov 2022 22:04:01 GMT
server: uvicorn
```

There are "Copy" and "Download" buttons next to the response body.

You can do the same to test DELETE `/events`, just make sure you give it whatever name you set for the event in your previous test of the PUT method.

### 4.1.2.5 Wrapping up

Now you have a fully functioning events API in FastAPI backed by EdgeDB. If you want to see all the source code for the completed project, you'll find it in [our examples repo](#). If you're stuck or if you just want to show off what you've built, come talk to us [on Discord](#). It's a great community of helpful folks, all passionate about being part of the next generation of databases.

If you like what you see and want to dive deeper into EdgeDB and what it can do, check out our [Easy EdgeDB book](#). In it, you'll get to learn more about EdgeDB as we build an imaginary role-playing game based on Bram Stoker's Dracula.

## 4.1.3 Flask

### edb-alt-title Building a REST API with EdgeDB and Flask

The EdgeDB Python client makes it easy to integrate EdgeDB into your preferred web development stack. In this tutorial, we'll see how you can quickly start building RESTful APIs with [Flask](#) and EdgeDB.

We'll build a simple movie organization system where you'll be able to fetch, create, update, and delete *movies* and *movie actors* via RESTful API endpoints.

### 4.1.3.1 Prerequisites

Before we start, make sure you've [installed](#) the `edgedb` command-line tool. Here, we'll use Python 3.10 and a few of its latest features while building the APIs. A working version of this tutorial can be found [on Github](#).

#### 4.1.3.1.1 Install the dependencies

To follow along, clone the repository and head over to the `flask-crud` directory.

```
$ git clone git@github.com:edgedb/edgedb-examples.git  
$ cd edgedb-examples/flask-crud
```

Create a Python 3.10 virtual environment, activate it, and install the dependencies with this command:

```
$ python -m venv myvenv  
$ source myvenv/bin/activate  
$ pip install edgedb flask 'httpx[cli]'
```

#### 4.1.3.1.2 Initialize the database

Now, let's initialize an EdgeDB project. From the project's root directory:

```
$ edgedb project init  
Initializing project...  
  
Specify the name of EdgeDB instance to use with this project  
[default: flask_crud]:  
> flask_crud  
  
Do you want to start instance automatically on login? [y/n]  
> y  
Checking EdgeDB versions...
```

Once you've answered the prompts, a new EdgeDB instance called `flask_crud` will be created and started.

#### 4.1.3.1.3 Connect to the database

Let's test that we can connect to the newly started instance. To do so, run:

```
$ edgedb
```

You should be connected to the database instance and able to see a prompt similar to this:

```
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb>
```

You can start writing queries here. However, the database is currently empty. Let's start designing the data model.

#### 4.1.3.2 Schema design

The movie organization system will have two object types—**movies** and **actors**. Each *movie* can have links to multiple *actors*. The goal is to create API endpoints that'll allow us to fetch, create, update, and delete the objects while maintaining their relationships.

EdgeDB allows us to declaratively define the structure of the objects. The schema lives inside `.esdl` file in the `dbschema` directory. It's common to declare the entire schema in a single file `dbschema/default.esdl`. This is how our datatypes look:

```
# dbschema/default.esdl

module default {
    abstract type Auditable {
        property created_at -> datetime {
            readonly := true;
            default := datetime_current();
        }
    }

    type Actor extending Auditable {
        required property name -> str {
            constraint max_len_value(50);
        }
        property age -> int16 {
            constraint min_value(0);
            constraint max_value(100);
        }
        property height -> int16 {
            constraint min_value(0);
            constraint max_value(300);
        }
    }

    type Movie extending Auditable {
        required property name -> str {
            constraint max_len_value(50);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
  property year -> int16{
    constraint min_value(1850);
  };
  multi link actors -> Actor;
}
}

```

Here, we've defined an `abstract` type called `Auditable` to take advantage of EdgeDB's schema mixin system. This allows us to add a `created_at` property to multiple types without repeating ourselves.

The `Actor` type extends `Auditable` and inherits the `created_at` property as a result. This property is auto-filled via the `datetime_current` function. Along with the inherited type, the `actor` type also defines a few additional properties like called `name`, `age`, and `height`. The constraints on the properties make sure that actor names can't be longer than 50 characters, `age` must be between 0 to 100 years, and finally, `height` must be between 0 to 300 centimeters.

We also define a `Movie` type that extends the `Auditable` abstract type. It also contains some additional concrete properties and links: `name`, `year`, and an optional multi-link called `actors` which refers to the `Actor` objects.

#### 4.1.3.3 Build the API endpoints

The API endpoints are defined in the `app` directory. The directory structure looks as follows:

```

app
└── __init__.py
└── actors.py
└── main.py
└── movies.py

```

The `actors.py` and `movies.py` modules contain the code to build the `Actor` and `Movie` APIs respectively. The `main.py` module then registers all the endpoints and exposes them to the webserver.

##### 4.1.3.3.1 Fetch actors

Since the `Actor` type is simpler, we'll start with that. Let's create a `GET /actors` endpoint so that we can see the `Actor` objects saved in the database. You can create the API in Flask like this:

```

# flask-crud/app/actors.py
from __future__ import annotations

import json
from http import HTTPStatus

import edgedb
from flask import Blueprint, request

actor = Blueprint("actor", __name__)
client = edgedb.create_client()

@actor.route("/actors", methods=["GET"])
def get_actors() -> tuple[dict, int]:

```

(continues on next page)

(continued from previous page)

```

filter_name = request.args.get("filter_name")

if not filter_name:
    actors = client.query_json(
        """
        select Actor {
            name,
            age,
            height
        }
        """
    )
else:
    actors = client.query_json(
        """
        select Actor {
            name,
            age,
            height
        }
        filter .name = <str>$filter_name
        """,
        filter_name=filter_name,
    )

response_payload = {"result": json.loads(actors)}
return response_payload, HTTPStatus.OK

```

The Blueprint instance does the actual work of exposing the API. We also create a blocking EdgeDB client instance to communicate with the database. By default, this API will return a list of actors, but you can also filter the objects by name.

In the `get_actors` function, we perform the database query via the `edgedb` client. Here, the `client.query_json` method conveniently returns JSON serialized objects. We deserialize the returned data in the `response_payload` dictionary and then return it. Afterward, the final JSON serialization part is taken care of by Flask. This endpoint is exposed to the server in the `main.py` module. Here's the content of the module:

```

# flask-crud/app/main.py
from __future__ import annotations

from flask import Flask

from app.actors import actor
from app.movies import movie

app = Flask(__name__)

app.register_blueprint(actor)
app.register_blueprint(movie)

```

To test the endpoint, go to the `flask-crud` directory and run:

```
$ export FLASK_APP=app.main:app && flask run --reload
```

This will start the development server and make it accessible via port 5000. Earlier, we installed the [HTTPx](#) client library to make HTTP requests programmatically. It also comes with a neat command-line tool that we'll use to test our API.

While the development server is running, on a new console, run:

```
$ httpx -m GET http://localhost:5000/actors
```

You'll see the following output on the console:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.1.1 Python/3.10.4
Date: Wed, 27 Apr 2022 18:58:38 GMT
Content-Type: application/json
Content-Length: 2

{
    "result": []
}
```

Our request yielded an empty list because the database is currently empty. Let's create the POST /actors endpoint to start saving actors in the database.

#### 4.1.3.3.2 Create actor

The POST endpoint can be built similarly:

```
# flask-crud/app/actors.py
...
@actor.route("/actors", methods=["POST"])
def post_actor() -> tuple[dict, int]:
    incoming_payload = request.json

    # Data validation.
    if not incoming_payload:
        return {
            "error": "Bad request"
        }, HTTPStatus.BAD_REQUEST

    if not (name := incoming_payload.get("name")):
        return {
            "error": "Field 'name' is required."
        }, HTTPStatus.BAD_REQUEST

    if len(name) > 50:
        return {
            "error": "Field 'name' cannot be longer than 50 "
                     "characters."
        }, HTTPStatus.BAD_REQUEST

    if age := incoming_payload.get("age"):
```

(continues on next page)

(continued from previous page)

```

if 0 <= age <= 100:
    return {
        "error": "Field 'age' must be between 0 "
        "and 100."
    }, HTTPStatus.BAD_REQUEST

if height := incoming_payload.get("height"):
    if not 0 <= height <= 300:
        return {
            "error": "Field 'height' must between 0 and "
            "300 cm."
        }, HTTPStatus.BAD_REQUEST

# Create object.
actor = client.query_single_json(
    """
    with
        name := <str>$name,
        age := <optional int16>$age,
        height := <optional int16>$height
    select (
        insert Actor {
            name := name,
            age := age,
            height := height
        }
    ){ name, age, height };
    """,
    name=name,
    age=age,
    height=height,
)
response_payload = {"result": json.loads(actor)}
return response_payload, HTTPStatus.CREATED

```

In the above snippet, we perform data validation in the conditional blocks and then make the query to create the object in the database. For now, we'll only allow creating a single object per request. The `client.query_single_json` ensures that we're creating and returning only one object. Inside the query string, notice, how we're using `<optional type>` to deal with the optional fields. If the user doesn't provide the value of an optional field like `age` or `height`, it'll be defaulted to `null`.

To test it out, make a request as follows:

```
$ httpx -m POST http://localhost:5000/actors \
-j '{"name": "Robert Downey Jr."}'
```

The output should look similar to this:

```
HTTP/1.1 201 CREATED
...
{
    "result": {
```

(continues on next page)

(continued from previous page)

```
"age": null,  
"height": null,  
"name": "Robert Downey Jr."  
}  
}
```

Before we move on to the next step, create 2 more actors called Chris Evans and Natalie Portman. Now that we have some data in the database, let's make a GET request to see the objects:

```
$ httpx -m GET http://localhost:5000/actors
```

The response looks as follows:

```
HTTP/1.1 200 OK  
...  
{  
  "result": [  
    {  
      "age": null,  
      "height": null,  
      "name": "Robert Downey Jr."  
    },  
    {  
      "age": null,  
      "height": null,  
      "name": "Chris Evans"  
    },  
    {  
      "age": null,  
      "height": null,  
      "name": "Natalie Portman"  
    }  
  ]  
}
```

You can filter the output of the GET /actors by name. To do so, use the `filter_name` query parameter like this:

```
$ httpx -m GET http://localhost:5000/actors \  
-p filter_name "Robert Downey Jr."
```

Doing this will only display the data of a single object:

```
HTTP/1.1 200 OK  
{  
  "result": [  
    {  
      "age": null,  
      "height": null,  
      "name": "Robert Downey Jr."  
    }  
  ]  
}
```

(continues on next page)

(continued from previous page)

```
]
}
```

Once you've done that, we can move on to the next step of building the PUT /actors endpoint to update the actor data.

#### 4.1.3.3.3 Update actor

It can be built like this:

```
# flask-crud/app/actors.py

# ...

@actor.route("/actors", methods=["PUT"])
def put_actors() -> tuple[dict, int]:
    incoming_payload = request.json
    filter_name = request.args.get("filter_name")

    # Data validation.
    if not incoming_payload:
        return {
            "error": "Bad request"
        }, HTTPStatus.BAD_REQUEST

    if not filter_name:
        return {
            "error": "Query parameter 'filter_name' must "
            "be provided",
        }, HTTPStatus.BAD_REQUEST

    if (name := incoming_payload.get("name")) and len(name) > 50:
        return {
            "error": "Field 'name' cannot be longer than "
            "50 characters."
        }, HTTPStatus.BAD_REQUEST

    if age := incoming_payload.get("age"):
        if age <= 0:
            return {
                "error": "Field 'age' cannot be less than "
                "or equal to 0."
            }, HTTPStatus.BAD_REQUEST

    if height := incoming_payload.get("height"):
        if not 0 <= height <= 300:
            return {
                "error": "Field 'height' must between 0 "
                "and 300 cm."
            }, HTTPStatus.BAD_REQUEST

    # Update object.
```

(continues on next page)

(continued from previous page)

```

actors = client.query_json(
    """
    with
        filter_name := <str>$filter_name,
        name := <optional str>$name,
        age := <optional int16>$age,
        height := <optional int16>$height
    select (
        update Actor
        filter .name = filter_name
        set {
            name := name ?? .name,
            age := age ?? .age,
            height := height ?? .height
        }
    ){ name, age, height };"",
    filter_name=filter_name,
    name=name,
    age=age,
    height=height,
)
response_payload = {"result": json.loads(actors)}
return response_payload, HTTPStatus.OK

```

Here, we'll isolate the intended object that we want to update by filtering the actors with the `filter_name` parameter. For example, if you wanted to update the properties of Robert Downey Jr., the value of the `filter_name` query parameter would be `Robert Downey Jr.`. The coalesce operator `??` in the query string makes sure that the API user can selectively update the properties of the target object and the other properties keep their existing values.

The following command updates the age and height of Robert Downey Jr..

```
$ httpx -m PUT http://localhost:5000/actors \
    -p filter_name "Robert Downey Jr." \
    -j '{"age": 57, "height": 173}'
```

This will return:

```
HTTP/1.1 200 OK
...
{
    "result": [
        {
            "age": 57,
            "height": 173,
            "name": "Robert Downey Jr."
        }
    ]
}
```

#### 4.1.3.3.4 Delete actor

Another API that we'll need to cover is the `DELETE /actors` endpoint. It'll allow us to query the name of the targeted object and delete that. The code looks similar to the ones you've already seen:

```
# flask-crud/app/actors.py
...

@actor.route("/actors", methods=["DELETE"])
def delete_actors() -> tuple[dict, int]:
    if not (filter_name := request.args.get("filter_name")):
        return {
            "error": "Query parameter 'filter_name' must "
            "be provided",
        }, HTTPStatus.BAD_REQUEST

    try:
        actors = client.query_json(
            """select (
                delete Actor
                filter .name = <str>$filter_name
            ) {name}
            """,
            filter_name=filter_name,
        )
    except edgedb.errors.ConstraintViolationError:
        return {
            "error": f"Cannot delete '{filter_name}'. "
            "Actor is associated with at least one movie."
        },
        HTTPStatus.BAD_REQUEST,
    )

    response_payload = {"result": json.loads(actors)}
    return response_payload, HTTPStatus.OK
```

This endpoint will simply delete the requested actor if the actor isn't attached to any movie. If the targeted object is attached to a movie, then API will throw an HTTP 400 (bad request) error and refuse to delete the object. To delete `Natalie Portman`, on your console, run:

```
$ httpx -m DELETE http://localhost:5000/actors \
-p filter_name "Natalie Portman"
```

That'll return:

```
HTTP/1.1 200 OK
...
{
    "result": [
        {
            "name": "Natalie Portman"
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```

    }
]
}
}
```

Now let's move on to building the Movie API.

#### 4.1.3.3.5 Create movie

Here's how we'll implement the POST /movie endpoint:

```

# flask-crud/app/movies.py
from __future__ import annotations

import json
from http import HTTPStatus

import edgedb
from flask import Blueprint, request

movie = Blueprint("movie", __name__)
client = edgedb.create_client()

@movie.route("/movies", methods=["POST"])
def post_movie() -> tuple[dict, int]:
    incoming_payload = request.json

    # Data validation.
    if not incoming_payload:
        return {
            "error": "Bad request"
        }, HTTPStatus.BAD_REQUEST

    if not (name := incoming_payload.get("name")):
        return {
            "error": "Field 'name' is required."
        }, HTTPStatus.BAD_REQUEST

    if len(name) > 50:
        return {
            "error": "Field 'name' cannot be longer than "
                     "50 characters."
        }, HTTPStatus.BAD_REQUEST

    if year := incoming_payload.get("year"):
        if year < 1850:
            return {
                "error": "Field 'year' cannot be less "
                         "than 1850."
            }, HTTPStatus.BAD_REQUEST

    actor_names = incoming_payload.get("actor_names")
```

(continues on next page)

(continued from previous page)

```

# Create object.
movie = client.query_single_json(
    """
    with
        name := <str>$name,
        year := <optional int16>$year,
        actor_names := <optional array<str>>$actor_names
    select (
        insert Movie {
            name := name,
            year := year,
            actors := (
                select Actor
                filter .name in array_unpack(actor_names)
            )
        }
    ){ name, year, actors: {name, age, height} };
    """,
    name=name,
    year=year,
    actor_names=actor_names,
)
response_payload = {"result": json.loads(movie)}
return response_payload, HTTPStatus.CREATED

```

Like the POST `/actors` API, conditional blocks validate the shape of the incoming data and the `client.query_json` method creates the object in the database. EdgeQL allows us to perform insertion and selection of data fields at the same time in a single query. One thing that's different here is that the POST `/movies` API also accepts an optional field called `actor_names` where the user can provide an array of actor names. The backend will associate the actors with the movie object if those actors exist in the database.

Here's how you'd create a movie:

```
$ httpx -m POST http://localhost:5000/movies \
    -j '{ "name": "The Avengers", "year": 2012, "actor_names": [ "Robert Downey Jr.", \
    ↵ "Chris Evans" ] }'
```

That'll return:

```
HTTP/1.1 201 CREATED
...
{
    "result": {
        "actors": [
            {
                "age": null,
                "height": null,
                "name": "Chris Evans"
            },
            {
                "age": 57,
                "height": 173,
            }
        ]
    }
}
```

(continues on next page)

(continued from previous page)

```
        "name": "Robert Downey Jr."
    }
],
"name": "The Avengers",
"year": 2012
}
}
```

#### 4.1.3.3.6 Additional movie endpoints

The implementation of the GET `/movie`, PATCH `/movie` and DELETE `/movie` endpoints are provided in the sample codebase in `app/movies.py`. But try to write them on your own using the Actor endpoints as a starting point! Once you're done, you should be able to fetch a movie by its title from your database with the `filter_name` parameter and the GET API as follows:

```
$ httpx -m GET http://localhost:5000/movies \
    -p 'filter_name' 'The Avengers'
```

That'll return:

```
HTTP/1.1 200 OK
...
{
  "result": [
    {
      "actors": [
        {
          "age": null,
          "name": "Chris Evans"
        },
        {
          "age": 57,
          "name": "Robert Downey Jr."
        }
      ],
      "name": "The Avengers",
      "year": 2012
    }
  ]
}
```

#### 4.1.3.4 Conclusion

While building REST APIs, the EdgeDB client allows you to leverage EdgeDB with any microframework of your choice. Whether it's [FastAPI](#), [Flask](#), [AIOHTTP](#), [Starlette](#), or [Tornado](#), the core workflow is quite similar to the one demonstrated above; you'll query and serialize data with the client and then return the payload for your framework to process.

### 4.1.4 Phoenix

#### **edb-alt-title** Building a GitHub OAuth application

There is a community-supported [Elixir](#) driver for EdgeDB. In this tutorial, we'll look at how you can create an application with authorization through GitHub using [Phoenix](#) and EdgeDB.

This tutorial is a simplified version of the [LiveBeats](#) application from [fly.io](#) with EdgeDB instead of PostgreSQL, which focuses on implementing authorization via GitHub. The completed implementation of this example can be found [on GitHub](#). The full version of LiveBeats version on EdgeDB can also be found [on GitHub](#)

#### 4.1.4.1 Prerequisites

For this tutorial we will need:

- EdgeDB CLI.
- Elixir version 1.13 or higher.
- Phoenix framework version 1.6 or higher.
- [GitHub OAuth application](#).

Before discussing the project database schema, let's generate a skeleton for our application. We will make sure that it will use binary IDs for the Ecto schemas because EdgeDB uses UUIDs as primary IDs, which in Elixir are represented as strings, and since it is basically a plain JSON API application, we will disable all the built-in Phoenix integrations.

```
$ mix phx.new phoenix-github_oauth --app github_oauth --module GitHubOAuth \
>   --no-html --no-gettext --no-dashboard --no-live --no-mailer --binary-id
$ cd phoenix-github_oauth/
```

Let's also get rid of some default things that were created by Phoenix and won't be used by us.

```
$ # remove the module Ecto.Repo and the directory for Ecto migrations,
$ # because they will not be used
$ rm -r lib/github_oauth/repo.ex priv/repo/
```

And then add the EdgeDB driver, the Ecto helper for it and the Mint HTTP client for GitHub OAuth client as project dependencies to `mix.exs`.

```
defmodule GitHubOAuth.MixProject do
  # ...

  defp deps do
    [
      {:phoenix, "~> 1.6.9"},
      {:phoenix_ecto, "~> 4.4"},
      {:esbuild, "~> 0.4", runtime: Mix.env() == :dev},
      {:telemetry_metrics, "~> 0.6"},
```

(continues on next page)

(continued from previous page)

```

{:telemetry_poller, "~> 1.0"},
{:jason, "~> 1.2"},
{:plug_cowboy, "~> 2.5"},
{:edgedb, "~> 0.3.0"},
{:edgedb_ecto, git: "https://github.com/nsidnev/edgedb_ecto"},
{:mint, "~> 1.0"} # we need mint to write the GitHub client
]
end

# ...
end

```

Now we need to download new dependencies.

```
$ mix deps.get
```

Next, we will create a module in `lib/github_oauth/edgedb.ex` which will define a child specification for the EdgeDB driver and use the EdgeDBEcto helper, which will inspect the queries that will be stored in the `priv/edgeql/` directory and generate Elixir code for them.

```

defmodule GitHubOAuth.EdgeDB do
  use EdgeDBEcto,
    name: __MODULE__,
    queries: true,
    otp_app: :github_oauth

  def child_spec(_opts \\ []) do
    %{
      id: __MODULE__,
      start: {EdgeDB, :start_link, [[name: __MODULE__]]}}
    }
  end
end

```

Now we need to add `GitHubOAuth.EdgeDB` as a child for our application in `lib/github_oauth/application.ex` (at the same time removing the child definition for `Ecto.Repo` from there).

```

defmodule GitHubOAuth.Application do
  # ...

  @impl true
  def start(_type, _args) do
    children = [
      # Start the EdgeDB driver
      GitHubOAuth.EdgeDB,
      # Start the Telemetry supervisor
      GitHubOAuthWeb.Telemetry,
      # Start the PubSub system
      {Phoenix.PubSub, name: GitHubOAuth.PubSub},
      # Start the Endpoint (http/https)
      GitHubOAuthWeb.Endpoint
      # Start a worker by calling: GitHubOAuth.Worker.start_link(arg)
      # {GitHubOAuth.Worker, arg}
    ]
  end
end

```

(continues on next page)

(continued from previous page)

```
]
# ...
end

# ...
end
```

Now we are ready to start working with EdgeDB! First, let's initialize a new project for this application.

```
$ edgedb project init
No `edgedb.toml` found in `/home/<user>/phoenix-github_oauth` or above

Do you want to initialize a new project? [Y/n]
> Y

Specify the name of EdgeDB instance to use with this project
[default: phoenix_github_oauth]:
> github_oauth

Checking EdgeDB versions...
Specify the version of EdgeDB to use with this project [default: 2.x]:
> 2.x

Do you want to start instance automatically on login? [y/n]
> y
```

Great! Now we are ready to develop the database schema for the application.

#### 4.1.4.2 Schema design

This application will have 2 types: `User` and `Identity`. The `default::User` represents the system user and the `default::Identity` represents the way the user logs in to the application (in this example via GitHub OAuth).

This schema will be stored in a single EdgeDB module inside the `dbschema/default.esdl` file.

```
module default {
    type User {
        property name -> str;
        required property username -> str;
        required property email -> cistr;

        property profile_tagline -> str;

        property avatar_url -> str;
        property external_homepage_url -> str;

        required property inserted_at -> cal::local_datetime {
            default := cal::to_local_datetime(datetime_current(), 'UTC');
        }

        required property updated_at -> cal::local_datetime {
```

(continues on next page)

(continued from previous page)

```

    default := cal::to_local_datetime(datetime_current(), 'UTC');
}

index on (.email);
index on (.username);
}

type Identity {
    required property provider -> str;
    required property provider_token -> str;
    required property provider_login -> str;
    required property provider_email -> str;
    required property provider_id -> str;

    required property provider_meta -> json {
        default := <json>"{}";
    }

    required property inserted_at -> cal::local_datetime {
        default := cal::to_local_datetime(datetime_current(), 'UTC');
    }

    required property updated_at -> cal::local_datetime {
        default := cal::to_local_datetime(datetime_current(), 'UTC');
    }

    required link user -> User {
        on target delete delete source;
    }

    index on (.provider);
    constraint exclusive on ((.user, .provider));
}
}

```

After saving the file, we can create a migration for the schema and apply the generated migration.

```

$ edgedb migration create
did you create object type 'default::User'? [y,n,l,c,b,s,q,?]
> y

did you create object type 'default::Identity'? [y,n,l,c,b,s,q,?]
> y

Created ./dbschema/migrations/00001.edgeql, id:
mlyehm3jhj6jqwguelek54jzp4wqvqqgrcnvcxwb7676ult7nmcta

$ edgedb migrate

```

#### 4.1.4.3 Ecto schemas

In this tutorial we will define 2 Ecto.Schema modules, for default::User and default::Identity types, so that we can work with EdgeDB in a more convenient way that is familiar to the world of Elixir.

Here is the definition for the user in the lib/accounts/user.ex file.

```
defmodule GitHubOAuth.Accounts.User do
  use Ecto.Schema
  use EdgeDBEcto.Mapper

  alias GitHubOAuth.Accounts.Identity

  @primary_key {:id, :binary_id, autogenerate: false}

  schema "default::User" do
    field :email, :string
    field :name, :string
    field :username, :string
    field :avatar_url, :string
    field :external_homepage_url, :string

    has_many :identities, Identity

    timestamps()
  end
end
```

And here for identity in lib/accounts/identity.ex.

```
defmodule GitHubOAuth.Accounts.Identity do
  use Ecto.Schema
  use EdgeDBEcto.Mapper

  alias GitHubOAuth.Accounts.User

  @primary_key {:id, :binary_id, autogenerate: false}

  schema "default::Identity" do
    field :provider, :string
    field :provider_token, :string
    field :provider_email, :string
    field :provider_login, :string
    field :provider_name, :string, virtual: true
    field :provider_id, :string
    field :provider_meta, :map

    belongs_to :user, User

    timestamps()
  end
end
```

#### 4.1.4.4 User authentication via GitHub

This part will be pretty big, as we'll talk about using `Ecto.Changeset` with the EdgeDB driver, as well as modules and queries related to user registration via GitHub OAuth.

`Ecto` provides “changesets” (via `Ecto.Changeset`), which are convenient to use when working with `Ecto.Schema` to validate external parameters. We could use them via `EdgeDBEcto` instead, though not quite as fully as we can with the full-featured adapters for `Ecto`.

First, we will update the `GitHubOAuth.Accounts.Identity` module so that it checks all the necessary parameters when we are creating a user via a GitHub registration.

```
defmodule GitHubOAuth.Accounts.Identity do
  # ...
  import Ecto.Changeset

  alias GitHubOAuth.Accounts.{Identity, User}

  @github "github"

  # ...

  def github_registration_changeset(info, primary_email, emails, token) do
    params = %{
      "provider_token" => token,
      "provider_id" => to_string(info["id"]),
      "provider_login" => info["login"],
      "provider_name" => info["name"] || info["login"],
      "provider_email" => primary_email
    }

    %Identity{}
    |> cast(params, [
      :provider_token,
      :provider_email,
      :provider_login,
      :provider_name,
      :provider_id
    ])
    |> put_change(:provider, @github)
    |> put_change(:provider_meta, %{"user" => info, "emails" => emails})
    |> validate_required([
      :provider_token,
      :provider_email,
      :provider_name,
      :provider_id
    ])
  end
end
```

And now let's define a changeset for user registration, which will use an already defined changeset from `GitHubOAuth.Accounts.Identity`.

```
defmodule GitHubOAuth.Accounts.User do
  # ...
```

(continues on next page)

(continued from previous page)

```

import Ecto.Changeset

alias GitHubOAuth.Accounts.{User, Identity}

# ...

def github_registration_changeset(info, primary_email, emails, token) do
  %{
    "login" => username,
    "avatar_url" => avatar_url,
    "html_url" => external_homepage_url
  } = info

  identity_changeset =
    Identity.github_registration_changeset(
      info,
      primary_email,
      emails,
      token
    )

  if identity_changeset.valid? do
    params = %{
      "username" => username,
      "email" => primary_email,
      "name" => get_change(identity_changeset, :provider_name),
      "avatar_url" => avatar_url,
      "external_homepage_url" => external_homepage_url
    }

    %User{}
    |> cast(params, [
      :email,
      :name,
      :username,
      :avatar_url,
      :external_homepage_url
    ])
    |> validate_required([:email, :name, :username])
    |> validate_username()
    |> validate_email()
    |> put_assoc(:identities, [identity_changeset])
  else
    %User{}
    |> change()
    |> Map.put(:valid?, false)
    |> put_assoc(:identities, [identity_changeset])
  end
end

defp validate_email(changeset) do

```

(continues on next page)

(continued from previous page)

```

changeset
|> validate_required(:email)
|> validate_format(
  :email,
  ~r/^[\s]+@[^\s]+\$/,
  message: "must have the @ sign and no spaces"
)
|> validate_length(:email, max: 160)
end

defp validate_username(changeset) do
  validate_format(changeset, :username, ~r/^[\w-]{2,32}$/)
end
end

```

Now that we have the schemas and changesets defined, let's define a set of the EdgeQL queries we need for the login process.

There are 5 queries that we will need:

1. Search for a user by user ID.
2. Search for a user by email and by identity provider.
3. Update the identity token if the user from the 1st query exists.
4. Registering a user along with his identity data, if the 1st request did not return the user.
5. Querying a user identity before updating its token.

Before writing the queries themselves, let's create a context module `lib/github_oauth/accounts.ex` that will use these queries, and the module itself will be used by Phoenix controllers.

```

defmodule GitHubOAuth.Accounts do
  import Ecto.Changeset

  alias GitHubOAuth.Accounts.{User, Identity}

  def get_user(id) do
    GitHubOAuth.EdgeDB.Accounts.get_user_by_id(id: id)
  end

  def register_github_user(primary_email, info, emails, token) do
    if user = get_user_by_provider(:github, primary_email) do
      update_github_token(user, token)
    else
      info
      |> User.github_registration_changeset(primary_email, emails, token)
      |> EdgeDBEcto.insert(
        &GitHubOAuth.EdgeDB.Accounts.register_github_user/1,
        nested: true
      )
    end
  end

  def get_user_by_provider(provider, email) when provider in [:github] do

```

(continues on next page)

(continued from previous page)

```

GitHubOAuth.EdgeDB.Accounts.get_user_by_provider(
    provider: to_string(provider),
    email: String.downcase(email)
)
end

defp update_github_token(%User{} = user, new_token) do
    identity =
        GitHubOAuth.EdgeDB.Accounts.get_identity_for_user(
            user_id: user.id,
            provider: "github"
        )

    {:ok, _} =
        identity
        |> change()
        |> put_change(:provider_token, new_token)
        |> EdgeDBEcto.update(
            &GitHubOAuth.EdgeDB.Accounts.update_identity_token/1
        )

    identity = %Identity{identity | provider_token: new_token}
    {:ok, %User{user | identities: [identity]}}
end
end

```

Note that updating a token with a single query is quite easy, but we will use two separate queries, to show how to work with `Ecto.Changeset` in different ways.

Now that all the preparations are complete, we can start writing EdgeQL queries.

We start with the `priv/edgeql/accounts/get_user_by_provider.edgeql` file, which defines a query to find an user with a specified email provider.

```

# edgedb = :query_single!
# mapper = GitHubOAuth.Accounts.User

select User {
    id,
    name,
    username,
    email,
    avatar_url,
    external_homepage_url,
    inserted_at,
    updated_at,
}
filter
    .<user[is Identity].provider = <str>$provider
    and
    str_lower(.email) = str_lower(<str>$email)
limit 1

```

It is worth noting the `# edgedb = :query_single!` and `# mapper = GitHubOAuth.Accounts.User` comments.

Both are special comments that will be used by EdgeDBEcto when generating query functions. The `edgedb` comment defines the driver function for requesting data. Information on all supported features can be found in the driver [documentation](#). The `mapper` comment is used to define the module that will be used to map the result from EdgeDB to some other form. Our `Ecto.Schema` schemas support this with use `EdgeDBEcto.Mapper` expression at the top of the module definition.

The queries for [getting the identity](#) and [getting the user by ID](#) are quite similar to the above, so we will omit them here. You can find these queries in the [example repository](#).

Instead, let's look at how to update the user identity. This will be described in the `priv/edgeql/accounts/update_identity_token.edgeql` file.

```
# edgedb = :query_required_single

with params := <json>$params
update Identity
filter .id = <uuid>params["id"]
set {
    provider_token := (
        <str>json_get(params, "provider_token") ?? .provider_token
    ),
    updated_at := cal::to_local_datetime(datetime_current(), 'UTC'),
}
```

As you can see, this query uses the named parameter `$params` instead of two separate parameters such as `$id` and `$provider_token`. This is because to update our identity we use the changeset in the module `GitHubOAuth.Accounts`, which automatically monitors changes to the schema and will not give back the parameters, which will not affect the state of the schema in update. So EdgeDBEcto automatically converts data from changesets when it is an update or insert operation into a named `$params` parameter of type JSON. It also helps to work with nested changesets, as we will see in the next query, which is defined in the `priv/edgeql/accounts/register_github_user.edgeql` file.

```
# edgedb = :query_single!
# mapper = GitHubOAuth.Accounts.User

with
    params := <json>$params,
    identities_params := params["identities"],
    user := (
        insert User {
            email := <str>params["email"],
            name := <str>params["name"],
            username := <str>params["username"],
            avatar_url := <optional str>json_get(params, "avatar_url"),
            external_homepage_url := (
                <str>json_get(params, "external_homepage_url")
            ),
        }
    ),
    identities := (
        for identity_params in json_array_unpack(identities_params) union (
            insert Identity {
                provider := <str>identity_params["provider"],
                provider_token := <str>identity_params["provider_token"],
                provider_email := <str>identity_params["provider_email"],
            }
        )
    )
)
```

(continues on next page)

(continued from previous page)

```

        provider_login := <str>identity_params["provider_login"],
        provider_id := <str>identity_params["provider_id"],
        provider_meta := <json>identity_params["provider_meta"],
        user := user,
    }
)
)
select user {
    id,
    name,
    username,
    email,
    avatar_url,
    external_homepage_url,
    inserted_at,
    updated_at,
    identities := identites,
}

```

Awesome! We're almost done with our application!

As a final step in this tutorial, we will add 2 routes for the web application. The first will redirect the user to the GitHub OAuth page if they're not already logged in or will show their username otherwise. The second is for logging into the application through GitHub.

Save the GitHub OAuth credentials from the *prerequisites* step as GITHUB\_CLIENT\_ID and GITHUB\_CLIENT\_SECRET environment variables.

And then modify your config/dev.exs configuration file to use them.

```

# ...

config :github_oauth, :github,
  client_id: System.fetch_env!("GITHUB_CLIENT_ID"),
  client_secret: System.fetch_env!("GITHUB_CLIENT_SECRET")

# ...

```

First we create a file lib/github\_oauth\_web/controllers/user\_controller.ex with a controller which will show the name of the logged in user or redirect to the authentication page otherwise.

```

defmodule GitHubOAuthWeb.UserController do
  use GitHubOAuthWeb, :controller

  alias GitHubOAuth.Accounts

  plug :fetch_current_user

  def index(conn, _params) do
    if conn.assigns.current_user do
      json(conn, %{name: conn.assigns.current_user.name})
    else
      redirect(conn, external: GitHubOAuth.GitHub.authorize_url())
    end
  end

```

(continues on next page)

(continued from previous page)

```

end

defp fetch_current_user(conn, _opts) do
  user_id = get_session(conn, :user_id)
  user = user_id && Accounts.get_user(user_id)
  assign(conn, :current_user, user)
end
end

```

Note that the implementation of the `GitHubOAuth.GitHub` module is not given here because it is relatively big and not a necessary part of this guide. If you want to explore its internals, you can check out its implementation on [GitHub](#).

Now add an authentication controller in `lib/github_oauth_web/controllers/oauth_callback_controller.ex`.

```

defmodule GitHubOAuthWeb.OAuthCallbackController do
  use GitHubOAuthWeb, :controller

  alias GitHubOAuth.Accounts

  require Logger

  def new(
    conn,
    %{"provider" => "github", "code" => code, "state" => state}
  ) do
    client = github_client(conn)

    with {:ok, info} <- client.exchange_access_token(code: code, state: state),
    %{
      info: info,
      primary_email: primary,
      emails: emails,
      token: token
    } = info,
    {:ok, user} <- Accounts.register_github_user(primary, info, emails, token) do
      conn
      |> log_in_user(user)
      |> redirect(to: "/")
    else
      {:error, %Ecto.Changeset{} = changeset} ->
        Logger.debug("failed GitHub insert #{inspect(changeset.errors)}")

      error =
        "We were unable to fetch the necessary information from " <>
        "your GitHub account"

      json(conn, %{error: error})

      {:error, reason} ->
        Logger.debug("failed GitHub exchange #{inspect(reason)}")
    end
  end
end

```

(continues on next page)

(continued from previous page)

```

        json(conn, %{
            error: "We were unable to contact GitHub. Please try again later"
        })
    end
end

def new(conn, %{"provider" => "github", "error" => "access_denied"}) do
    json(conn, %{error: "Access denied"})
end

defp github_client(conn) do
    conn.assigns[:github_client] || GitHubOAuth.GitHub
end

defp log_in_user(conn, user) do
    conn
    |> assign(:current_user, user)
    |> configure_session(renew: true)
    |> clear_session()
    |> put_session(:user_id, user.id)
end
end

```

Finally, we need to change `lib/github_oauth_web/router.ex` and add new controllers there.

```

defmodule GitHubOAuthWeb.Router do
    # ...

    pipeline :api do
        # ...
        plug :fetch_session
    end

    scope "/", GitHubOAuthWeb do
        pipe_through :api

        get "/", UserController, :index
        get "/oauth/callbacks/:provider", OAuthCallbackController, :new
    end

    # ...
end

```

#### 4.1.4.5 Running web server

That's it! Now we are ready to run our application and check if everything works as expected.

```
$ mix phx.server
Generated github_oauth app
[info] Running GitHubOAuthWeb.Endpoint with cowboy 2.9.0 at 127.0.0.1:4000
(HTTP)

[info] Access GitHubOAuthWeb.Endpoint at http://localhost:4000
```

After going to <http://localhost:4000>, we will be greeted by the GitHub authentication page. And after confirming the login we will be automatically redirected back to our local server, which will save the received user in the session and return the obtained user name in the JSON response.

We can also verify that everything is saved correctly by manually checking the database data.

```
edgedb> select User {
.....   name,
.....   username,
.....   avatar_url,
.....   external_homepage_url,
..... };
{
  default::User {
    name: 'Nik',
    username: 'nsidnev',
    avatar_url: 'https://avatars.githubusercontent.com/u/22559461?v=4',
    external_homepage_url: 'https://github.com/nsidnev'
  },
}
edgedb> select Identity {
.....   provider,
.....   provider_login
..... }
..... filter .user.username = 'nsidnev';
{default::Identity {provider: 'github', provider_login: 'nsidnev'}}}
```

#### 4.1.5 Strawberry

**edb-alt-title** Building a GraphQL API with EdgeDB and Strawberry

EdgeDB allows you to query your database with GraphQL via the built-in GraphQL extension. It enables you to expose GraphQL-driven CRUD APIs for all object types, their properties, links, and aliases. This opens up the scope for creating backend-less applications where the users will directly communicate with the database. You can learn more about that in the [GraphQL](#) section of the docs.

However, as of now, EdgeDB is not ready to be used as a standalone backend. You shouldn't expose your EdgeDB instance directly to the application's frontend; this is insecure and will give all users full read/write access to your database. So, in this tutorial, we'll see how you can quickly create a simple GraphQL API without using the built-in extension, which will give the users restricted access to the database schema. Also, we'll implement HTTP basic authentication and demonstrate how you can write your own GraphQL validators and resolvers. This tutorial assumes you're already familiar with GraphQL terms like schema, query, mutation, resolver, validator, etc, and have used GraphQL with some other technology before.

We'll build the same movie organization system that we used in the Flask [tutorial](#) and expose the objects and relationships as a GraphQL API. Using the GraphQL interface, you'll be able to fetch, create, update, and delete movie and actor objects in the database. [Strawberry](#) is a Python library that takes a code-first approach where you'll write your object schema as Python classes. This allows us to focus more on how you can integrate EdgeDB into your workflow and less on the idiosyncrasies of GraphQL itself. We'll also use the EdgeDB client to communicate with the database, [FastAPI](#) to build the authentication layer, and Uvicorn as the webserver.

#### 4.1.5.1 Prerequisites

Before we start, make sure you have *installed* the `edgedb` command-line tool. Here, we'll use Python 3.10 and a few of its latest features while building the APIs. A working version of this tutorial can be found [on Github](#).

##### 4.1.5.1.1 Install the dependencies

To follow along, clone the repository and head over to the `strawberry-gql` directory.

```
$ git clone git@github.com:edgedb/edgedb-examples.git  
$ cd edgedb-examples/strawberry-gql
```

Create a Python 3.10 virtual environment, activate it, and install the dependencies with this command:

```
$ python3.10 -m venv .venv  
$ source .venv/bin/activate  
$ pip install edgedb fastapi strawberry-graphql uvicorn[standard]
```

##### 4.1.5.1.2 Initialize the database

Now, let's initialize an EdgeDB project. From the project's root directory:

```
$ edgedb project init  
Initializing project...  
  
Specify the name of EdgeDB instance to use with this project  
[default: strawberry_crud]:  
> strawberry_crud  
  
Do you want to start instance automatically on login? [y/n]  
> y  
Checking EdgeDB versions...
```

Once you've answered the prompts, a new EdgeDB instance called `strawberry_crud` will be created and started.

#### 4.1.5.1.3 Connect to the database

Let's test that we can connect to the newly started instance. To do so, run:

```
$ edgedb
```

You should be connected to the database instance and able to see a prompt similar to this:

```
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb>
```

You can start writing queries here. However, the database is currently empty. Let's start designing the data model.

#### 4.1.5.2 Schema design

The movie organization system will have two object types—**movies** and **actors**. Each *movie* can have links to multiple *actors*. The goal is to create a GraphQL API suite that'll allow us to fetch, create, update, and delete the objects while maintaining their relationships.

EdgeDB allows us to declaratively define the structure of the objects. The schema lives inside .esdl file in the dbschema directory. It's common to declare the entire schema in a single file dbschema/default.esdl. This is how our datatypes look:

```
# dbschema/default.esdl

module default {
    abstract type Auditable {
        property created_at -> datetime {
            readonly := true;
            default := datetime_current();
        }
    }

    type Actor extending Auditable {
        required property name -> str {
            constraint max_len_value(50);
        }
        property age -> int16 {
            constraint min_value(0);
            constraint max_value(100);
        }
        property height -> int16 {
            constraint min_value(0);
            constraint max_value(300);
        }
    }

    type Movie extending Auditable {
        required property name -> str {
            constraint max_len_value(50);
        }
        property year -> int16{
```

(continues on next page)

(continued from previous page)

```

    constraint min_value(1850);
};

multi link actors -> Actor;
}
}

```

Here, we've defined an abstract type called `Auditable` to take advantage of EdgeDB's schema mixin system. This allows us to add a `created_at` property to multiple types without repeating ourselves.

The `Actor` type extends `Auditable` and inherits the `created_at` property as a result. This property is auto-filled via the `datetime_current` function. Along with the inherited type, the actor type also defines a few additional properties like called `name`, `age`, and `height`. The constraints on the properties make sure that actor names can't be longer than 50 characters, age must be between 0 to 100 years, and finally, height must be between 0 to 300 centimeters.

We also define a `Movie` type that extends the `Auditable` abstract type. It also contains some additional concrete properties and links: `name`, `year`, and an optional multi-link called `actors` which refers to the `Actor` objects.

#### 4.1.5.3 Build the GraphQL API

The API endpoints are defined in the `app` directory. The directory structure looks as follows:

```

app
├── __init__.py
└── main.py
└── schemas.py

```

The `schemas.py` module contains the code that defines the GraphQL schema and builds the queries and mutations for `Actor` and `Movie` objects. The `main.py` module then registers the GraphQL schema, adds the authentication layer, and exposes the API to the webserver.

##### 4.1.5.3.1 Write the GraphQL schema

Along with the database schema, to expose EdgeDB's object relational model as a GraphQL API, you'll also have to define a GraphQL schema that mirrors the object structure in the database. Strawberry allows us to express this schema via type annotated Python classes. We define the Strawberry schema in the `schema.py` file as follows:

```

# strawberry-gql/app/schema.py
from __future__ import annotations

import json # will be used later for serialization

import edgedb
import strawberry

client = edgedb.create_async_client()

@strawberry.type
class Actor:
    name: str | None
    age: int | None = None
    height: int | None = None

```

(continues on next page)

(continued from previous page)

```
@strawberry.type
class Movie:
    name: str | None
    year: int | None = None
    actors: list[Actor] | None = None
```

Here, the GraphQL schema mimics our database schema. Similar to the `Actor` and `Movie` types in the EdgeDB schema, here, both the `Actor` and `Movie` models have three attributes. Likewise, the `actors` attribute in the `Movie` model represents the link between movies and actors.

#### 4.1.5.3.2 Query actors

In this section, we'll write the resolver to create the queries that'll allow us to fetch the actor objects from the database. You'll need to write the query resolvers as methods in a class decorated with the `@strawberry.type` decorator. Each method will also need to be decorated with the `@strawberry.field` decorator to mark them as resolvers. Resolvers can be either sync or async. In this particular case, we'll write asynchronous resolvers that'll act in a non-blocking manner. The query to fetch the actors is built in the `schema.py` file as follows:

```
# strawberry-gql/app/schema.py
...

@strawberry.type
class Query:
    @strawberry.field
    @async def get_actors(
        self, filter_name: str | None = None
    ) -> list[Actor]:
        if filter_name:
            actors_json = await client.query_json(
                """
                    select Actor {name, age, height}
                    filter .name=<str>$filter_name
                """,
                filter_name=filter_name,
            )
        else:
            actors_json = await client.query_json(
                """
                    select Actor {name, age, height}
                """
            )
        actors = json.loads(actors_json)
        return [
            Actor(name, age, height)
            for (name, age, height) in (
                d.values() for d in actors
            )
        ]
```

(continues on next page)

(continued from previous page)

```
# Register the Query.
schema = strawberry.Schema(query=Query)
```

Here, the `get_actors` resolver method accepts an optional `filter_name` parameter and returns a list of `Actor` type objects. The optional `filter_name` parameter allows us to build the capability of filtering the actor objects by name. Inside the method, we use the EdgeDB client to asynchronously query the data. The `client.query_json` method returns JSON serialized data which we use to create the `Actor` instances. Finally, we return the list of actor instances and the rest of the work is done by Strawberry. Then in the last line of the above snippet, we register the `Query` class to build the `Schema` instance.

Afterward, in the `main.py` module, we use FastAPI to expose the `/graphql` endpoint. Also, we add a basic HTTP authentication layer to demonstrate how you can easily protect your GraphQL endpoint by leveraging FastAPI's dependency injection system. Here's how the content of the `main.py` looks:

```
# strawberry-gql/app/main.py
from __future__ import annotations

import secrets
from typing import Literal

from fastapi import (
    Depends, FastAPI, HTTPException, Request,
    Response, status
)
from fastapi.security import HTTPBasic, HTTPBasicCredentials
from strawberry.fastapi import GraphQLRouter

from app.schema import schema

app = FastAPI()
router = GraphQLRouter(schema)
security = HTTPBasic()

def auth(
    credentials: HTTPBasicCredentials = Depends(security)
) -> Literal[True]:
    """Simple HTTP Basic Auth."""

    correct_username = secrets.compare_digest(
        credentials.username, "ubuntu"
    )
    correct_password = secrets.compare_digest(
        credentials.password, "debian"
    )

    if not (correct_username and correct_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Basic"},
        )
```

(continues on next page)

(continued from previous page)

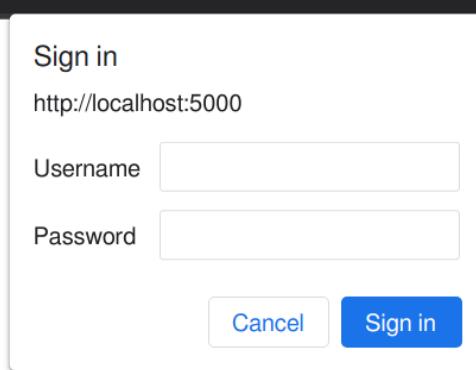
```
)  
return True  
  
@router.api_route("/", methods=["GET", "POST"])  
async def graphql(request: Request) -> Response:  
    return await router.handle_graphql(request=request)  
  
app.include_router(  
    router, prefix="/graphql", dependencies=[Depends(auth)]  
)
```

First, we initialize the `FastAPI` app instance which will communicate with the Uvicorn webserver. Then we attach the initialized `schema` instance to the `GraphQLRouter`. The `HTTPBasic` class provides the machinery required to add the authentication layer. The `auth` function houses the implementation details of how we're comparing the incoming and expected username and passwords as well as how the webserver is going to handle unauthorized requests. The `graphql` handler function is the one that handles the incoming HTTP requests. Finally, the `router` instance and the security handler are registered to the `app` instance via the `app.include_router` method.

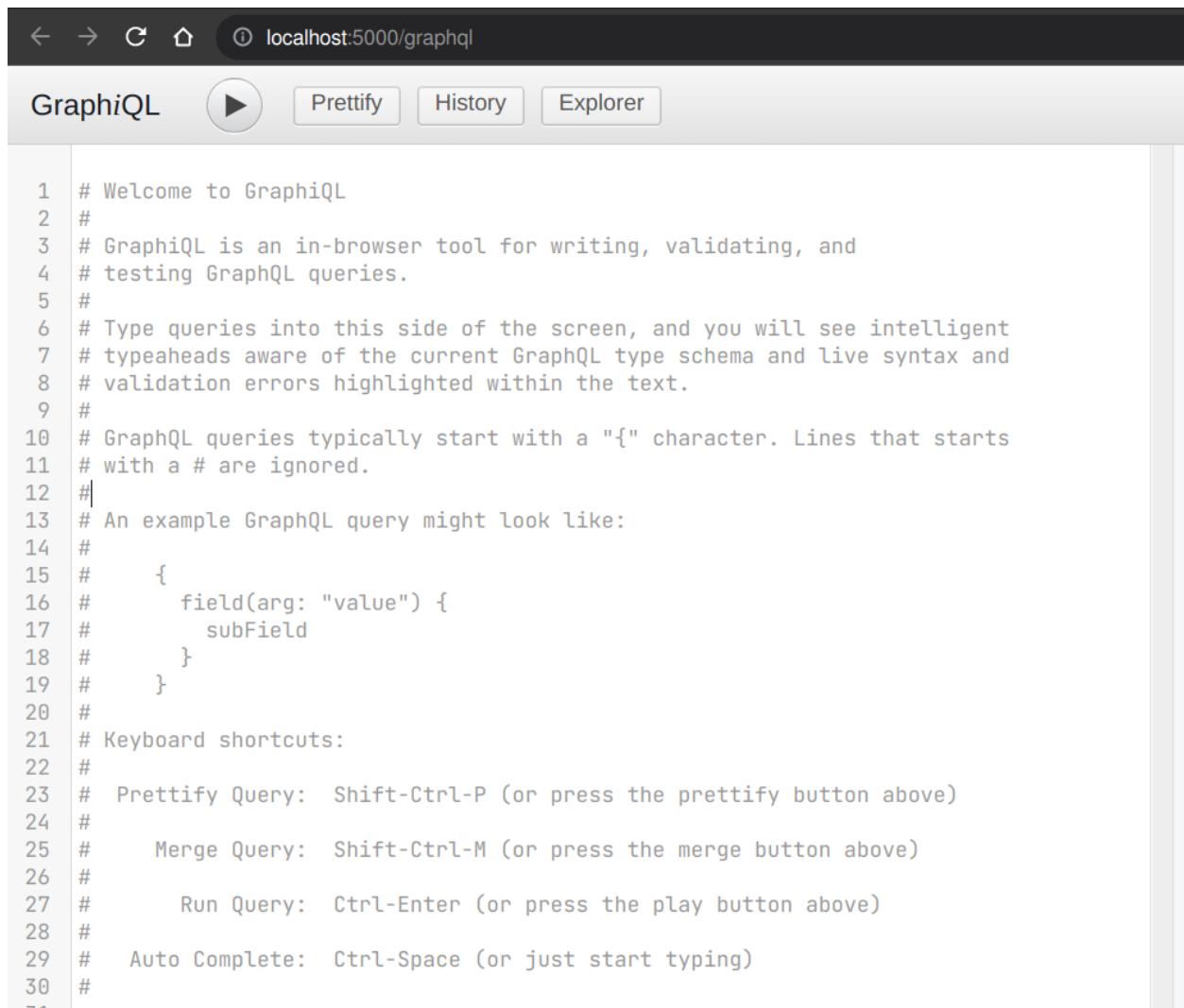
We can now start querying the `/graphql` endpoint. We'll use the built-in GraphiQL interface to perform the queries. Before that, let's start the Uvicorn webserver first. Run:

```
$ uvicorn app.main:app --port 5000 --reload
```

This exposes the webserver in port 5000. Now, in your browser, go to <http://localhost:5000/graphql>. Here, you'll find that the HTTP basic auth requires us to provide the username and password.



Currently, the allowed username and password is `ubuntu` and `debian` respectively. Provide the credentials and you'll be taken to a page that looks like this:



The screenshot shows the GraphiQL interface running at `localhost:5000/graphql`. The title bar says "GraphiQL". Below it are tabs for "Prettify", "History", and "Explorer". The main area contains a multi-line text block with line numbers from 1 to 30. The text is a welcome message and a list of keyboard shortcuts:

```

1 # Welcome to GraphiQL
2 #
3 # GraphiQL is an in-browser tool for writing, validating, and
4 # testing GraphQL queries.
5 #
6 # Type queries into this side of the screen, and you will see intelligent
7 # typeaheads aware of the current GraphQL type schema and live syntax and
8 # validation errors highlighted within the text.
9 #
10 # GraphQL queries typically start with a "{" character. Lines that starts
11 # with a # are ignored.
12 #
13 # An example GraphQL query might look like:
14 #
15 #     {
16 #         field(arg: "value") {
17 #             subField
18 #         }
19 #     }
20 #
21 # Keyboard shortcuts:
22 #
23 # Prettify Query: Shift-Ctrl-P (or press the prettify button above)
24 #
25 # Merge Query: Shift-Ctrl-M (or press the merge button above)
26 #
27 # Run Query: Ctrl-Enter (or press the play button above)
28 #
29 # Auto Complete: Ctrl-Space (or just start typing)
30 #
z1

```

You can write your GraphQL queries here. Let's write a query that'll fetch all the actors in the database and show all three of their attributes. The following query does that:

```
query ActorQuery {
    getActors {
        age
        height
        name
    }
}
```

The following response will appear on the right panel of the GraphiQL explorer:

The screenshot shows a GraphQL interface with the URL `localhost:5000/graphql`. The query entered is:

```

1 query ActorQuery {
2   getActors{
3     age
4     height
5     name
6   }
7 }
8

```

The response payload is:

```

{
  "data": {
    "getActors": []
  }
}

```

Since as of now, the database doesn't have any data, the payload is returning an empty list. Let's write a mutation and create some actors.

#### 4.1.5.3.3 Mutate actors

Mutations are also written in the `schema.py` file. To write a mutation, you'll have to create a separate class where you'll write the mutation resolvers. The resolver methods will need to be decorated with the `@strawberry.mutation` decorator. You can write the mutation that'll create an actor object in the database as follows:

```
# strawberry-gql/app/schema.py
...

@strawberry.type
class Mutation:
    @strawberry.mutation
    async def create_actor(
        self, name: str,
        age: int | None = None,
        height: int | None = None
    ) -> ResponseActor:

        actor_json = await client.query_single_json(
            """
            with new_actor := (
                insert Actor {
                    name := <str>$name,
                    age := <optional int16>$age,
                    height := <optional int16>$height
                }
            )
                select new_actor {name, age, height}
            """
        )
        return ResponseActor(**actor_json)
```

(continues on next page)

(continued from previous page)

```
        name=name,
        age=age,
        height=height,
    )

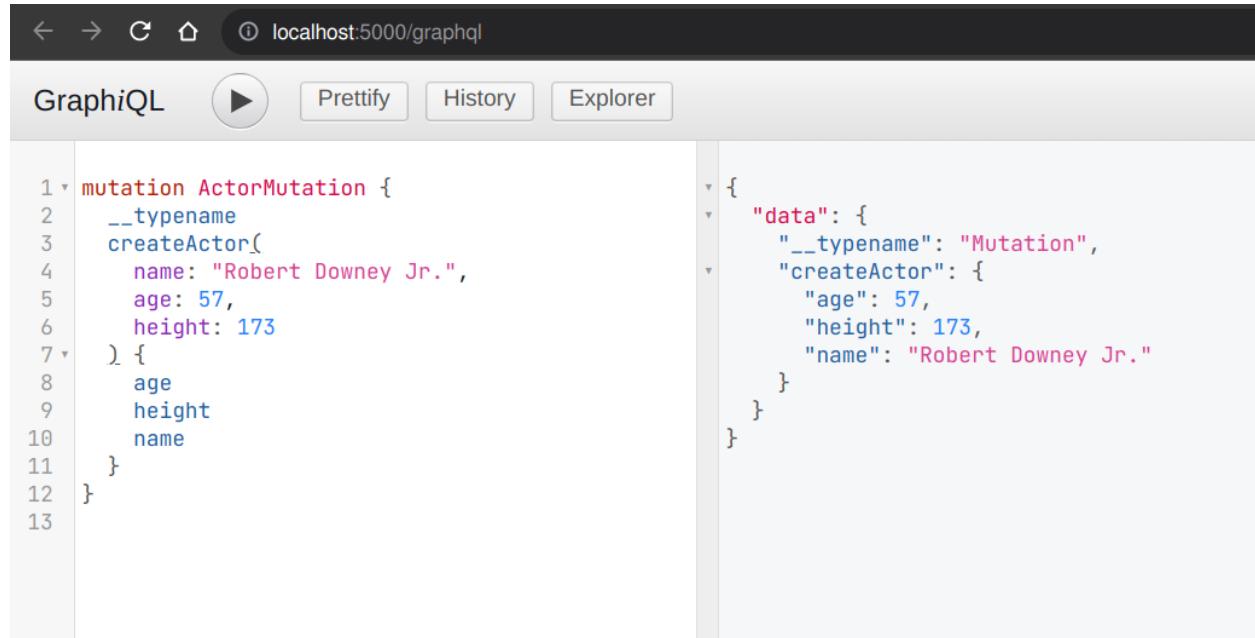
    actor = json.loads(actor_json)
    return Actor(
        actor.get("name"),
        actor.get("age"),
        actor.get("height")
    )

# Mutation class needs to be registered here.
schema = strawberry.Schema(query=Query, mutation=Mutation)
```

Creating a mutation also includes data validation. By type annotating the `Mutation` class, we're implicitly asking Strawberry to perform data validation on the incoming request payload. Strawberry will raise an HTTP 400 error if the validation fails. Let's create an actor. Submit the following GraphQL query in the GraphQL interface:

```
mutation ActorMutation {
    __typename
    createActor(
        name: "Robert Downey Jr.",
        age: 57,
        height: 173
    ) {
        age
        height
        name
    }
}
```

In the above mutation, `name` is a required field and the remaining two are optional fields. This mutation will create an actor named `Robert Downey Jr.` and show all three attributes—`name`, `age`, and `height` of the created actor in the response payload. Here's the response:



The screenshot shows the GraphiQL interface running at `localhost:5000/graphql`. The query editor contains the following GraphQL mutation:

```

1 mutation ActorMutation {
2   __typename
3   createActor(
4     name: "Robert Downey Jr.",
5     age: 57,
6     height: 173
7   ) {
8     age
9     height
10    name
11  }
12}
13

```

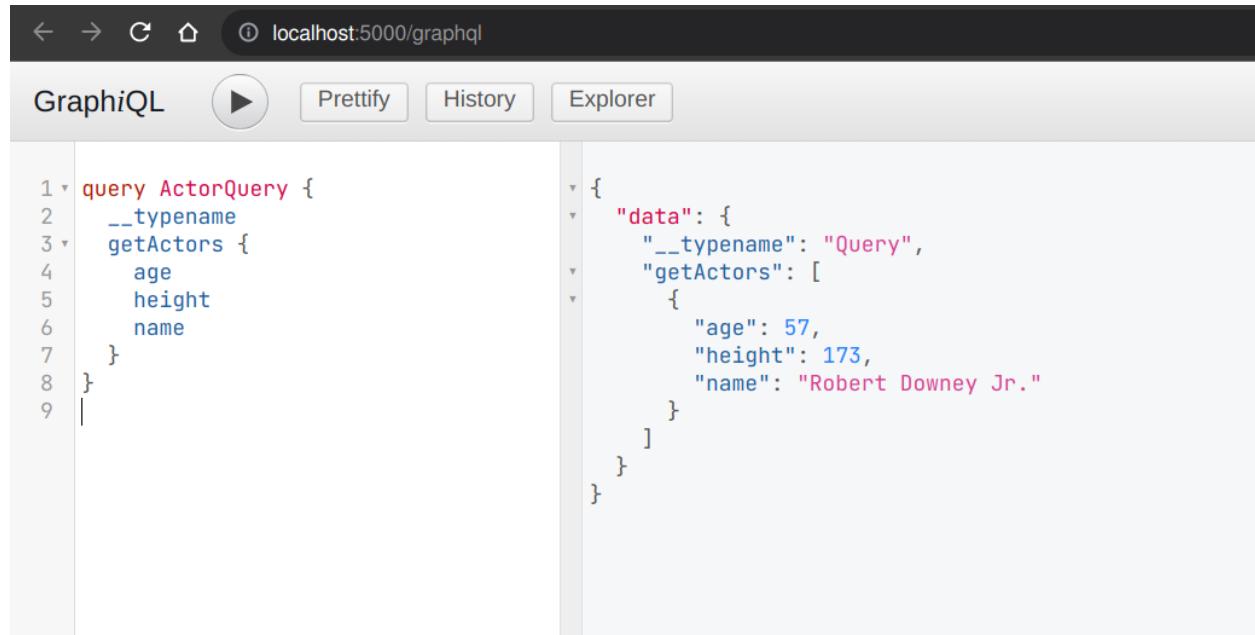
The response pane on the right shows the JSON result of the mutation:

```

{
  "data": {
    "__typename": "Mutation",
    "createActor": {
      "age": 57,
      "height": 173,
      "name": "Robert Downey Jr."
    }
  }
}

```

Now that we've created an actor object, we can run the previously created query to fetch the actors. Running the `ActorQuery` will give you the following response:



The screenshot shows the GraphiQL interface running at `localhost:5000/graphql`. The query editor contains the following GraphQL query:

```

1 query ActorQuery {
2   __typename
3   getActors {
4     age
5     height
6     name
7   }
8 }
9

```

The response pane on the right shows the JSON result of the query:

```

{
  "data": {
    "__typename": "Query",
    "getActors": [
      {
        "age": 57,
        "height": 173,
        "name": "Robert Downey Jr."
      }
    ]
  }
}

```

You can also filter actors by their names. To do so, you'd leverage the `filterName` parameter of the `getActors` resolver:

```

query ActorQuery {
  __typename
  getActors(filterName: "Robert Downey Jr.") {
    age
    height
    name
  }
}

```

(continues on next page)

(continued from previous page)

```

    }
}
```

This will only display the filtered results. Similarly, as shown above, you can write the mutations to update and delete actors. Their implementations can be found in the `schema.py` file. Check out `update_actors` and `delete_resolvers` to learn more about their implementation details. You can update one or more attributes of an actor with the following mutation:

```

mutation ActorMutation {
    __typename
    updateActors(filterName: "Robert Downey Jr.", age: 60) {
        name
        age
        height
    }
}
```

Running this mutation will update the `age` of Robert Downey Jr.. First, we filter the objects that we want to mutate via the `filterName` parameter and then we update the relevant attributes; in this case, we updated the `age` of the object. Finally, we show all the fields in the return payload. Use the GraphQL explorer to interactively play with the full API suite.

#### 4.1.5.3.4 Query movies

In the `schema.py` file, the query to fetch movies is constructed as follows:

```

# strawberry-gql/app/schema.py
...

@strawberry.type
class Query:
    ...

    @strawberry.field
    async def get_movies(
        self, filter_name: str | None = None,
    ) -> list[Movie]:
        if filter_name:
            movies_json = await client.query_json(
                """
                    select Movie {name, year, actors: {name, age, height}}
                    filter .name=<str>$filter_name
                """,
                filter_name=filter_name,
            )
        else:
            movies_json = await client.query_json(
                """
                    select Movie {name, year, actors: {name, age, height}}
                """
            )

    @strawberry.field
    def get_movie(self, movie_id: int) -> Movie:
        return await client.get_movie(movie_id)
```

(continues on next page)

(continued from previous page)

```

        )

    # Deserialize.
    movies = json.loads(movies_json)
    for idx, movie in enumerate(movies):
        actors = [
            Actor(name) for d in movie.get("actors", [])
            for name in d.values()
        ]

        movies[idx] = Movie(
            movie.get("name"),
            movie.get("year"), actors
        )
    return movies

```

Similar to the actor query, this also allows you to either fetch all or filter movies by the movie names. Execute the following query to see the movies in the database:

```

query MovieQuery {
    __typename
    getMovies {
        actors {
            age
            height
            name
        }
        name
        year
    }
}

```

This will return an empty list since the database doesn't contain any movies. In the next section, we'll create a mutation to create the movies and query them afterward.

#### 4.1.5.3.5 Mutate movies

Before running any query to fetch the movies, let's see how you'd construct a mutation that allows you to create movies. You can build the mutation similar to how we've constructed the create actor mutation. It looks like this:

```

# strawberry-gql/app/schema.py
...

@strawberry.type
class Mutation:
    ...

    @strawberry.mutation
    async def create_movie(
        self,
        name: str,
    ):
        ...

```

(continues on next page)

(continued from previous page)

```

year: int | None = None,
actor_names: list[str] | None = None,
) -> Movie:
    movie_json = await client.query_single_json(
        """
        with
            name := <str>$name,
            year := <optional int16>$year,
            actor_names := <optional array<str>>$actor_names,
            new_movie := (
                insert Movie {
                    name := name,
                    year := year,
                    actors := (
                        select detached Actor
                        filter .name in array_unpack(actor_names)
                    )
                }
            )
        select new_movie {
            name,
            year,
            actors: {name, age, height}
        }
        """,
        name=name,
        year=year,
        actor_names=actor_names,
    )

    movie = json.loads(movie_json)
    actors = [
        Actor(name) for d in movie.get("actors", [])
        for name in d.values()
    ]

    return Movie(
        movie.get("name"),
        movie.get("year"),
        actors
    )

```

You can submit a request to this mutation to create a movie. While creating a movie, you must provide the name of the movie as it's a required field. Also, you can optionally provide the year the movie was released and an array containing the names of the actors. If the values of the `actor_names` field match any existing actor in the database, the above snippet makes sure that the movie will be linked with the corresponding actors. In the GraphQL explorer, run the following mutation to create a movie named `Avengers` and link the actor `Robert Downey Jr.` with the movie:

```

mutation MovieMutation {
    __typename
    createMovie(
        name: "Avengers",
        actorNames: ["Robert Downey Jr."],

```

(continues on next page)

(continued from previous page)

```
    year: 2012
) {
    actors {
        name
    }
}
```

It'll return:

The screenshot shows a GraphQL interface with a query editor and a results panel. The query editor contains the following mutation:

```
1 mutation MovieMutation {
2     __typename
3     createMovie(
4         name: "Avengers",
5         actorNames: ["Robert Downey Jr."],
6         year: 2012
7     ) {
8         actors {
9             name
10        }
11    }
12 }
```

The results panel shows the JSON response:

```
{ "data": { "createMovie": { "actors": [ { "name": "Robert Downey Jr." } ] } }}
```

Now you can fetch the movies with a simple query like this one:

```
query MovieQuery {
    __typename
    getMovies {
        name
        year
        actors {
            name
        }
    }
}
```

You'll then see an output similar to this:

The screenshot shows a GraphQL playground interface. The URL in the address bar is `localhost:5000/graphql`. The top navigation bar includes 'GraphiQL' (selected), 'Prettify', 'History', and 'Explorer'. The left pane contains a code editor with the following GraphQL query:

```

1 query MovieQuery {
2   __typename
3   getMovies {
4     name
5     year
6     actors {
7       name
8     }
9   }
10 }
11

```

The right pane displays the JSON response from the server:

```

{
  "data": {
    "__typename": "Query",
    "getMovies": [
      {
        "name": "Avengers",
        "year": 2012,
        "actors": [
          {
            "name": "Robert Downey Jr."
          }
        ]
      }
    ]
  }
}

```

Take a look at the `update_movies` and `delete_movies` resolvers to gain more insights into the implementation details of those mutations.

#### 4.1.5.4 Conclusion

In this tutorial, you've seen how can use Strawberry and EdgeDB together to quickly build a fully-featured GraphQL API. Also, you have seen how FastAPI allows you add an authentication layer and serve the API in a secure manner. One thing to keep in mind here is, ideally, you'd only use GraphQL if you're interfacing with something that already expects a GraphQL API. Otherwise, EdgeQL is always going to be more powerful and expressive than GraphQL's query syntax.

## 4.2 Deployment

EdgeDB can be hosted on all major cloud hosting platforms. The guides below demonstrate how to spin up both a managed PostgreSQL instance and a container running EdgeDB [in Docker](#).

**Note:** Minimum requirements

As a rule of thumb, the EdgeDB Docker container requires 1GB RAM! Images with insufficient RAM may experience unexpected issues during startup.

## 4.2.1 AWS

### edb-alt-title Deploying EdgeDB to AWS

In this guide we show how to deploy EdgeDB on AWS using Amazon Aurora and Elastic Container Service.

#### 4.2.1.1 Prerequisites

- AWS account with billing enabled (or a free trial)
- (optional) aws CLI ([install](#))

#### 4.2.1.2 Quick Install with CloudFormation

We maintain a [CloudFormation template](#) for easy automated deployment of EdgeDB in your AWS account. The template deploys EdgeDB to a new ECS service and connects it to a newly provisioned Aurora PostgreSQL cluster. The created instance has a public IP address with TLS configured and is protected by a password you provide.

##### 4.2.1.2.1 CloudFormation Web Portal

Click [here](#) to start the deployment process using CloudFormation portal and follow the prompts. You'll be prompted to provide a value for the following parameters:

- DockerImage: defaults to the latest version (edgedb/edgedb), or you can specify a particular tag from the ones published to [Docker Hub](#).
- InstanceName: Due to limitations with AWS, this must be 22 characters or less!
- SuperUserPassword: this will used as the password for the new EdgeDB instance. Keep track of the value you provide.

Once the deployment is complete, follow these steps to find the host name that has been assigned to your EdgeDB instance:

1. Open the AWS Console and navigate to CloudFormation > Stacks. Click on the newly created Stack.
2. Wait for the status to read CREATE\_COMPLETE—it can take 15 minutes or more.
3. Once deployment is complete, click the Outputs tab. The value of PublicHostname is the hostname at which your EdgeDB instance is publicly available.
4. Copy the hostname and run the following command to open a REPL to your instance.

```
$ edgedb --dsn edgedb://edgedb:<password>@<hostname> --tls-security insecure
EdgeDB 2.x
Type \help for help, \quit to quit.
edgedb>
```

It's often convenient to create an alias for the remote instance using `edgedb instance link`.

```
$ edgedb instance link \
--trust-tls-cert \
--dsn edgedb://edgedb:<password>@<hostname>
my_aws_instance
```

This aliases the remote instance to `my_aws_instance` (this name can be anything). You can now use the `-I my_aws_instance` flag to run CLI commands against this instance, as with local instances.

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

```
$ edgedb -I my_aws_instance
EdgeDB 2.x
Type \help for help, \quit to quit.
edgedb>
```

#### 4.2.1.2.2 CloudFormation CLI

Alternatively, if you prefer to use AWS CLI, run the following command in your terminal:

```
$ aws cloudformation create-stack \
--stack-name EdgeDB \
--template-url \
  https://edgedb-deploy.s3.us-east-2.amazonaws.com/edgedb-aurora.yml \
--capabilities CAPABILITY_NAMED_IAM \
--parameters ParameterKey=SuperUserPassword,ParameterValue=<password>
```

#### 4.2.1.3 Manual Install with CLI

The following instructions produce a deployment that is very similar to the CloudFormation option above.

##### 4.2.1.3.1 Create a VPC

For convenience, assign a deployment name and region to environment variables. The `NAME` variable will be used as prefix for all the resources created throughout the process. It should only contain alphanumeric characters and hyphens.

```
$ NAME=your-deployment-name
$ REGION=us-west-2
```

Then create the VPC.

```
$ VPC_ID=$( \
  aws ec2 create-vpc \
  --region $REGION \
  --output text \
  --query "Vpc.VpcId" \
  --cidr-block "10.0.0.0/16" \
  --instance-tenancy default \
  --tag-specifications \
    "ResourceType=vpc,Tags=[{Key=Name,Value=${NAME}-vpc}]" \
)
$ aws ec2 modify-vpc-attribute \
  --region $REGION \
```

(continues on next page)

(continued from previous page)

```
--vpc-id $VPC_ID \
--enable-dns-support

$ aws ec2 modify-vpc-attribute \
--region $REGION \
--vpc-id $VPC_ID \
--enable-dns-hostnames
```

#### 4.2.1.3.2 Create a Gateway

Allow communication between the VPC and the internet by creating an Internet Gateway.

```
$ GATEWAY_ID=$( \
aws ec2 create-internet-gateway \
--region $REGION \
--output text \
--query "InternetGateway.InternetGatewayId" \
--tag-specifications \
"ResourceType=internet-gateway, \
Tags=[{Key=Name,Value=${NAME}-internet-gateway}]" \
)

$ aws ec2 attach-internet-gateway \
--region $REGION \
--internet-gateway-id $GATEWAY_ID \
--vpc-id $VPC_ID
```

#### 4.2.1.3.3 Create a Public Network ACL

A Network Access Control List will act as a firewall for a publicly accessible subnet.

```
$ PUBLIC_ACL_ID=$( \
aws ec2 create-network-acl \
--region $REGION \
--output text \
--query "NetworkAcl.NetworkAclId" \
--vpc-id $VPC_ID \
--tag-specifications \
"ResourceType=network-acl, \
Tags=[{Key=Name,Value=${NAME}-public-network-acl}]" \
)

$ aws ec2 create-network-acl-entry \
--region $REGION \
--network-acl-id $PUBLIC_ACL_ID \
--rule-number 99 \
--protocol 6 \
--port-range From=0,To=65535 \
--rule-action allow \
```

(continues on next page)

(continued from previous page)

```
--ingress \
--cidr-block 0.0.0.0/0

$ aws ec2 create-network-acl-entry \
--region $REGION \
--network-acl-id $PUBLIC_ACL_ID \
--rule-number 99 \
--protocol 6 \
--port-range From=0,To=65535 \
--rule-action allow \
--egress \
--cidr-block 0.0.0.0/0
```

#### 4.2.1.3.4 Create a Private Network ACL

A second ACL will be the firewall for a private subnet to provide an extra boundary around the PostgreSQL cluster.

```
$ PRIVATE_ACL_ID=$( \
aws ec2 create-network-acl \
--region $REGION \
--output text \
--query "NetworkAcl.NetworkAclId" \
--vpc-id $VPC_ID \
--tag-specifications \
"ResourceType=network-acl, \
Tags=[{Key=Name,Value=${NAME}-private-network-acl}]" \
)"

$ aws ec2 create-network-acl-entry \
--region $REGION \
--network-acl-id $PRIVATE_ACL_ID \
--rule-number 99 \
--protocol -1 \
--rule-action allow \
--ingress \
--cidr-block 0.0.0.0/0

$ aws ec2 create-network-acl-entry \
--region $REGION \
--network-acl-id $PRIVATE_ACL_ID \
--rule-number 99 \
--protocol -1 \
--rule-action allow \
--egress \
--cidr-block 0.0.0.0/0
```

#### 4.2.1.3.5 Create a Public Subnet in Availability Zone “A”

```
$ AVAILABILITY_ZONE_A=$( \
    aws ec2 describe-availability-zones \
    --region $REGION \
    --output text \
    --query "AvailabilityZones[0].ZoneName" \
)

$ SUBNET_A_PUBLIC_ID=$( \
    aws ec2 create-subnet \
    --region $REGION \
    --output text \
    --query "Subnet.SubnetId" \
    --availability-zone $AVAILABILITY_ZONE_A \
    --cidr-block 10.0.0.0/20 \
    --vpc-id $VPC_ID \
    --tag-specifications \
        "ResourceType=subnet, \
        Tags=[{Key=Name,Value=${NAME}-subnet-a-public}, \
              {Key=Reach,Value=public}]" \
)
)

$ aws ec2 replace-network-acl-association \
    --region $REGION \
    --network-acl-id $PUBLIC_ACL_ID \
    --association-id $( \
        aws ec2 describe-network-acls \
        --region $REGION \
        --output text \
        --query " \
            NetworkAccls[*].Associations[?SubnetId=='${SUBNET_A_PUBLIC_ID}'][] \
            | [0].NetworkAclAssociationId" \
    )
)

$ ROUTE_TABLE_A_PUBLIC_ID=$( \
    aws ec2 create-route-table \
    --region $REGION \
    --output text \
    --query "RouteTable.RouteTableId" \
    --vpc-id $VPC_ID \
    --tag-specifications \
        "ResourceType=route-table, \
        Tags=[{Key=Name,Value=${NAME}-route-table-a-public}]" \
)
)

$ aws ec2 create-route \
    --region $REGION \
    --route-table-id $ROUTE_TABLE_A_PUBLIC_ID \
    --destination-cidr-block 0.0.0.0/0 \
    --gateway-id $GATEWAY_ID
```

(continues on next page)

(continued from previous page)

```
$ aws ec2 associate-route-table \
--region $REGION \
--route-table-id $ROUTE_TABLE_A_PUBLIC_ID \
--subnet-id $SUBNET_A_PUBLIC_ID
```

#### 4.2.1.3.6 Create a Private Subnet in Availability Zone “A”

```
$ SUBNET_A_PRIVATE_ID=$( \
aws ec2 create-subnet \
--region $REGION \
--output text \
--query "Subnet.SubnetId" \
--availability-zone $AVAILABILITY_ZONE_A \
--cidr-block 10.0.16.0/20 \
--vpc-id $VPC_ID \
--tag-specifications \
"ResourceType=subnet, \
Tags=[{Key=Name,Value=${NAME}-subnet-a-private}, \
{Key=Reach,Value=private}]" \
)

$ aws ec2 replace-network-acl-association \
--region $REGION \
--network-acl-id $PRIVATE_ACL_ID \
--association-id ${ \
aws ec2 describe-network-acls \
--region $REGION \
--output text \
--query " \
NetworkAcls[*].Associations[?SubnetId == '${SUBNET_A_PRIVATE_ID}' \
][[] | [0].NetworkAclAssociationId" \
}

$ ROUTE_TABLE_A_PRIVATE_ID=$( \
aws ec2 create-route-table \
--region $REGION \
--output text \
--query "RouteTable.RouteTableId" \
--vpc-id $VPC_ID \
--tag-specifications \
"ResourceType=route-table, \
Tags=[{Key=Name,Value=${NAME}-route-table-a-private}]" \
)

$ aws ec2 associate-route-table \
--region $REGION \
--route-table-id $ROUTE_TABLE_A_PRIVATE_ID \
--subnet-id $SUBNET_A_PRIVATE_ID
```

#### 4.2.1.3.7 Create a Public Subnet in Availability Zone “B”

```
$ AVAILABILITY_ZONE_B=$( \
    aws ec2 describe-availability-zones \
    --region $REGION \
    --output text \
    --query "AvailabilityZones[1].ZoneName" \
)

$ SUBNET_B_PUBLIC_ID=$( \
    aws ec2 create-subnet \
    --region $REGION \
    --output text \
    --query "Subnet.SubnetId" \
    --availability-zone $AVAILABILITY_ZONE_B \
    --cidr-block 10.0.32.0/20 \
    --vpc-id $VPC_ID \
    --tag-specifications \
        "ResourceType=subnet, \
        Tags=[{Key=Name,Value=${NAME}-subnet-b-public}, \
              {Key=Reach,Value=public}]" \
)
)

$ aws ec2 replace-network-acl-association \
    --region $REGION \
    --network-acl-id $PUBLIC_ACL_ID \
    --association-id $( \
        aws ec2 describe-network-acls \
        --region $REGION \
        --output text \
        --query " \
            NetworkAccls[*].Associations[?SubnetId == '${SUBNET_B_PUBLIC_ID}' \
                ][] | [0].NetworkAclAssociationId" \
    )
)

$ ROUTE_TABLE_B_PUBLIC_ID=$( \
    aws ec2 create-route-table \
    --region $REGION \
    --output text \
    --query "RouteTable.RouteTableId" \
    --vpc-id $VPC_ID \
    --tag-specifications \
        "ResourceType=route-table, \
        Tags=[{Key=Name,Value=${NAME}-route-table-b-public}]" \
)
)

$ aws ec2 create-route \
    --region $REGION \
    --route-table-id $ROUTE_TABLE_B_PUBLIC_ID \
    --destination-cidr-block 0.0.0.0/0 \
    --gateway-id $GATEWAY_ID
```

(continues on next page)

(continued from previous page)

```
$ aws ec2 associate-route-table \
--region $REGION \
--route-table-id $ROUTE_TABLE_B_PUBLIC_ID \
--subnet-id $SUBNET_B_PUBLIC_ID
```

#### 4.2.1.3.8 Create a Private Subnet in Availability Zone “B”

```
$ SUBNET_B_PRIVATE_ID=$( \
aws ec2 create-subnet \
--region $REGION \
--output text \
--query "Subnet.SubnetId" \
--availability-zone $AVAILABILITY_ZONE_B \
--cidr-block 10.0.48.0/20 \
--vpc-id $VPC_ID \
--tag-specifications \
"ResourceType=subnet, \
Tags=[{Key=Name,Value=${NAME}-subnet-b-private}, \
{Key=Reach,Value=private}]" \
)

$ aws ec2 replace-network-acl-association \
--region $REGION \
--network-acl-id $PRIVATE_ACL_ID \
--association-id ${ \
aws ec2 describe-network-acls \
--region $REGION \
--output text \
--query " \
NetworkAcls[*].Associations[?SubnetId=='${SUBNET_B_PRIVATE_ID}'][] \
| [0].NetworkAclAssociationId" \
}

$ ROUTE_TABLE_B_PRIVATE_ID=$( \
aws ec2 create-route-table \
--region $REGION \
--output text \
--query "RouteTable.RouteTableId" \
--vpc-id $VPC_ID \
--tag-specifications \
"ResourceType=route-table, \
Tags=[{Key=Name,Value=${NAME}-route-table-b-private}]" \
)

$ aws ec2 associate-route-table \
--region $REGION \
--route-table-id $ROUTE_TABLE_B_PRIVATE_ID \
--subnet-id $SUBNET_B_PRIVATE_ID
```

#### 4.2.1.3.9 Create an EC2 security group

```
$ EC2_SECURITY_GROUP_ID=$( \
    aws ec2 create-security-group \
    --region $REGION \
    --output text \
    --query "GroupId" \
    --group-name "${NAME}-ec2-security-group" \
    --description "Controls access to ${NAME} stack EC2 instances." \
    --vpc-id $VPC_ID \
    --tag-specifications \
    "ResourceType=security-group, \
    Tags=[{Key=Name,Value=${NAME}-ec2-security-group}]" \
)

$ aws ec2 authorize-security-group-ingress \
    --region $REGION \
    --group-id $EC2_SECURITY_GROUP_ID \
    --protocol tcp \
    --cidr 0.0.0.0/0 \
    --port 5656 \
    --tag-specifications \
    "ResourceType=security-group-rule, \
    Tags=[{Key=Name,Value=${NAME}-ec2-security-group-ingress}]"
```

#### 4.2.1.3.10 Create an RDS Security Group

```
$ RDS_SECURITY_GROUP_ID=$( \
    aws ec2 create-security-group \
    --region $REGION \
    --output text \
    --query "GroupId" \
    --group-name "${NAME}-rds-security-group" \
    --description "Controls access to ${NAME} stack RDS instances." \
    --vpc-id $VPC_ID \
    --tag-specifications \
    "ResourceType=security-group, \
    Tags=[{Key=Name,Value=${NAME}-rds-security-group}]" \
)

$ aws ec2 authorize-security-group-ingress \
    --region $REGION \
    --group-id $RDS_SECURITY_GROUP_ID \
    --protocol tcp \
    --source-group $EC2_SECURITY_GROUP_ID \
    --port 5432 \
    --tag-specifications \
    "ResourceType=security-group-rule, \
    Tags=[{Key=Name,Value=${NAME}-rds-security-group-ingress}]"

$ RDS_SUBNET_GROUP_NAME="${NAME}-rds-subnet-group"
```

(continues on next page)

(continued from previous page)

```
$ aws rds create-db-subnet-group \
--region $REGION \
--db-subnet-group-name "$RDS_SUBNET_GROUP_NAME" \
--db-subnet-group-description "EdgeDB RDS subnet group for ${NAME}" \
--subnet-ids $SUBNET_A_PRIVATE_ID $SUBNET_B_PRIVATE_ID
```

#### 4.2.1.3.11 Create an RDS Cluster

Use the `read` command to securely assign a value to the `PASSWORD` environment variable.

```
$ echo -n "> " && read -s PASSWORD
```

Then use this password to create an AWS `secret`.

```
$ PASSWORD_ARN=$( \
aws secretsmanager create-secret \
--region $REGION \
--output text \
--query "ARN" \
--name "${NAME}-password" \
--secret-string "$PASSWORD" \
)"

$ DB_CLUSTER_IDENTIFIER="${NAME}-postgres-cluster"

$ DB_CLUSTER_ADDRESS=$( \
aws rds create-db-cluster \
--region $REGION \
--output text \
--query "DBCluster.Endpoint" \
--engine aurora-postgresql \
--engine-version 13.4 \
--db-cluster-identifier "$DB_CLUSTER_IDENTIFIER" \
--db-subnet-group-name "$RDS_SUBNET_GROUP_NAME" \
--master-username postgres \
--master-user-password "$PASSWORD" \
--port 5432 \
--vpc-security-group-ids "$RDS_SECURITY_GROUP_ID" \
)

$ aws rds create-db-instance \
--region $REGION \
--availability-zone "$AVAILABILITY_ZONE_A" \
--engine "aurora-postgresql" \
--db-cluster-identifier "$DB_CLUSTER_IDENTIFIER" \
--db-instance-identifier "${NAME}-postgres-instance-a" \
--db-instance-class "db.t3.medium" \
--db-subnet-group-name "$RDS_SUBNET_GROUP_NAME"

$ aws rds create-db-instance \
```

(continues on next page)

(continued from previous page)

```
--region $REGION \
--availability-zone "$AVAILABILITY_ZONE_B" \
--engine "aurora-postgresql" \
--db-cluster-identifier "$DB_CLUSTER_IDENTIFIER" \
--db-instance-identifier "${NAME}-postgres-instance-b" \
--db-instance-class "db.t3.medium" \
--db-subnet-group-name "$RDS_SUBNET_GROUP_NAME"

$ DSN_ARN=$( \
    aws secretsmanager create-secret \
    --region $REGION \
    --output text \
    --query "ARN" \
    --name "${NAME}-backend-dsn" \
    --secret-string \
    "postgres://postgres:${PASSWORD}@${DB_CLUSTER_ADDRESS}:5432/postgres" \
)''
```

#### 4.2.1.3.12 Create a Load Balancer

Adding a load balancer will facilitate scaling the EdgeDB cluster.

```
$ TARGET_GROUP_ARN=$( \
    aws elbv2 create-target-group \
    --region $REGION \
    --output text \
    --query "TargetGroups[0].TargetGroupArn" \
    --health-check-interval-seconds 10 \
    --health-check-path "/server/status/ready" \
    --health-check-protocol HTTPS \
    --unhealthy-threshold-count 2 \
    --healthy-threshold-count 2 \
    --name "${NAME}-target-group" \
    --port 5656 \
    --protocol TCP \
    --target-type ip \
    --vpc-id $VPC_ID \
)''

$ LOAD_BALANCER_NAME="${NAME}-load-balancer"

$ LOAD_BALANCER_ARN=$( \
    aws elbv2 create-load-balancer \
    --region $REGION \
    --output text \
    --query "LoadBalancers[0].LoadBalancerArn" \
    --type network \
    --name "$LOAD_BALANCER_NAME" \
    --scheme internet-facing \
    --subnets "$SUBNET_A_PUBLIC_ID" "$SUBNET_B_PUBLIC_ID" \
)''
```

(continues on next page)

(continued from previous page)

```
$ aws elbv2 create-listener \
--region $REGION \
--default-actions \
'[{"TargetGroupArn": """$TARGET_GROUP_ARN""", "Type": "forward"}]' \
--load-balancer-arn "$LOAD_BALANCER_ARN" \
--port 5656 \
--protocol TCP
```

#### 4.2.1.3.13 Create an ECS Cluster

The only thing left to do is create and ECS cluster and deploy the EdgeDB container in it.

```
$ EXECUTION_ROLE_NAME="${NAME}-execution-role"

$ EXECUTION_ROLE_ARN=$( \
aws iam create-role \
--region $REGION \
--output text \
--query "Role.Arn" \
--role-name "$EXECUTION_ROLE_NAME" \
--assume-role-policy-document \
'{ \
    "Version": "2012-10-17", \
    "Statement": [{} \
        {"Effect": "Allow", \
        "Principal": {"Service": "ecs-tasks.amazonaws.com"}, \
        "Action": "sts:AssumeRole"} \
    ] \
}' \
)

$ SECRETS_ACCESS_POLICY_ARN=$( \
aws iam create-policy \
--region $REGION \
--output text \
--query "Policy.Arn" \
--policy-name "${NAME}-secrets-access-policy" \
--policy-document \
'{ \
    "Version": "2012-10-17", \
    "Statement": [{} \
        {"Effect": "Allow", \
        "Action": "secretsmanager:GetSecretValue", \
        "Resource": ["$PASSWORD_ARN", \
                    "$DSN_ARN"] \
    ] \
}' \
)
```

(continues on next page)

(continued from previous page)

```
$ aws iam attach-role-policy \
  --region $REGION \
  --role-name "$EXECUTION_ROLE_NAME" \
  --policy-arn \
  "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"

$ aws iam attach-role-policy \
  --region $REGION \
  --role-name "$EXECUTION_ROLE_NAME" \
  --policy-arn "$SECRETS_ACCESS_POLICY_ARN"

$ TASK_ROLE_ARN=$( \
  aws iam create-role \
  --region $REGION \
  --output text \
  --query "Role.Arn" \
  --role-name "${NAME}-task-role" \
  --assume-role-policy-document \
  "{ \
    \"Version\": \"2012-10-17\", \
    \"Statement\": [{ \
      \"Effect\": \"Allow\", \
      \"Principal\": {\"Service\": \"ecs-tasks.amazonaws.com\"}, \
      \"Action\": \"sts:AssumeRole\" \
    }] \
  }" \
)

$ LOG_GROUP_NAME="/ecs/edgedb/${NAME}"

$ aws logs create-log-group \
  --region $REGION \
  --log-group-name "$LOG_GROUP_NAME"

$ CLUSTER_NAME="${NAME}-server-cluster"

$ aws ecs create-cluster \
  --region $REGION \
  --cluster-name "$CLUSTER_NAME"

$ LOG_GROUP_ARN=$( \
  aws logs describe-log-groups \
  --region $REGION \
  --output text \
  --query "logGroups[0].arn" \
  --log-group-name-prefix "$LOG_GROUP_NAME" \
)

$ TASK_DEFINITION_ARN=$( \
  aws ecs register-task-definition \
  --region $REGION \
```

(continues on next page)

(continued from previous page)

```
--output text \
--query "taskDefinition.taskDefinitionArn" \
--requires-compatibilities "FARGATE" \
--network-mode "awsvpc" \
--execution-role-arn "$EXECUTION_ROLE_ARN" \
--task-role-arn "$TASK_ROLE_ARN" \
--family "${NAME}-task-definition" \
--cpu 1024 \
--memory 2GB \
--container-definitions \
"[{" \
  "\"name\": \"$NAME\", \
  "\"image\": \"edgedb/edgedb\", \
  "\"portMappings\": [{\"containerPort\": 5656}], \
  "\"command\": [\"edgedb-server\"], \
  "\"environment\": [{ \
    "\"name\": \"EDGEDB_SERVER_GENERATE_SELF_SIGNED_CERT\", \
    "\"value\": \"1\" \
  }], \
  "\"secrets\": [ \
    { \
      "\"name\": \"EDGEDB_SERVER_PASSWORD\", \
      "\"valueFrom\": \"$PASSWORD_ARN\" \
    }, \
    { \
      "\"name\": \"EDGEDB_SERVER_BACKEND_DSN\", \
      "\"valueFrom\": \"$DSN_ARN\" \
    } \
  ], \
  "\"logConfiguration\": { \
    "\"logDriver\": \"awslogs\", \
    "\"options\": { \
      "\"awslogs-region\": \"$REGION\", \
      "\"awslogs-group\": \"$LOG_GROUP_NAME\", \
      "\"awslogs-stream-prefix\": \"ecs\" \
    } \
  } \
}]" \
)"

$ aws ecs create-service \
  --region $REGION \
  --service-name "$NAME" \
  --cluster "$CLUSTER_NAME" \
  --task-definition "$TASK_DEFINITION_ARN" \
  --deployment-configuration \
  "minimumHealthyPercent=100,maximumPercent=200" \
  --desired-count 2 \
  --health-check-grace-period-seconds 120 \
  --launch-type FARGATE \
  --network-configuration \
  "awsvpcConfiguration={ \

```

(continues on next page)

(continued from previous page)

```
assignPublicIp=ENABLED, \
subnets=[$SUBNET_A_PUBLIC_ID,$SUBNET_B_PUBLIC_ID], \
securityGroups=[$EC2_SECURITY_GROUP_ID] \
}" \
--load-balancers \
"containerName=$NAME, \
containerPort=5656, \
targetGroupArn=$TARGET_GROUP_ARN"
```

#### 4.2.1.3.14 Create a local link to the new EdgeDB instance

Create an local alias to the remote EdgeDB instance with `edgedb instance link`:

```
$ printf $PASSWORD | edgedb instance link \
--password-from-stdin \
--trust-tls-cert \
--non-interactive \
--host "$(
aws ec2 describe-network-interfaces \
--output text \
--region $REGION \
--query \
"NetworkInterfaces[?contains(Description, '$LOAD_BALANCER_NAME')] \
| [0].Association.PublicIp" \
)" \
aws
```

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

You can now open a REPL to this instance

#### 4.2.1.4 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

## 4.2.2 Azure

**edb-alt-title** Deploying EdgeDB to Azure

In this guide we show how to deploy EdgeDB using Azure's Postgres Flexible Server as the backend.

#### 4.2.2.1 Prerequisites

- Valid Azure Subscription with billing enabled or credits ([free trial](#)).
- Azure CLI ([install](#)).

#### 4.2.2.2 Provision an EdgeDB instance

Login to your Microsoft Azure account.

```
$ az login
```

Create a new resource group.

```
$ GROUP=my-group-name
$ az group create --name $GROUP --location westus
```

Provision a PostgreSQL server.

---

**Note:** If you already have a database provisioned you can skip this step.

---

For convenience, assign a value to the PG\_SERVER\_NAME environment variable; we'll use this variable in multiple later commands.

```
$ PG_SERVER_NAME=postgres-for-edgedb
```

Use the read command to securely assign a value to the PASSWORD environment variable.

```
$ echo -n ">" && read -s PASSWORD
```

Then create a Postgres Flexible server.

```
$ az postgres flexible-server create \
  --resource-group $GROUP \
  --name $PG_SERVER_NAME \
  --location westus \
  --admin-user edgedb \
  --admin-password $PASSWORD \
  --sku-name Standard_D2s_v3 \
  --version 13 \
  --yes
```

---

**Note:** If you get an error saying "Specified server name is already used." change the value of PG\_SERVER\_NAME and rerun the command.

---

Allow other Azure services access to the Postgres instance.

```
$ az postgres flexible-server firewall-rule create \
  --resource-group $GROUP \
  --name $PG_SERVER_NAME \
  --rule-name allow-azure-internal \
```

(continues on next page)

(continued from previous page)

```
--start-ip-address 0.0.0.0 \
--end-ip-address 0.0.0.0
```

EdgeDB requires Postgres' `uuid-ossp` extension which needs to be enabled.

```
$ az postgres flexible-server parameter set \
  --resource-group $GROUP \
  --server-name $PG_SERVER_NAME \
  --name azure.extensions \
  --value uuid-ossp
```

Start an EdgeDB container.

```
$ PG_HOST=$(
  az postgres flexible-server list \
    --resource-group $GROUP \
    --query "[?name=='$PG_SERVER_NAME'].fullyQualifiedDomainName | [0]" \
    --output tsv
)
$ DSN="postgresql://edgedb:$PASSWORD@$PG_HOST/postgres?sslmode=require"
$ az container create \
  --resource-group $GROUP \
  --name edgedb-container-group \
  --image edgedb/edgedb \
  --dns-name-label edgedb \
  --ports 5656 \
  --secure-environment-variables \
  "EDGEDB_SERVER_PASSWORD=$PASSWORD" \
  "EDGEDB_SERVER_BACKEND_DSN=$DSN" \
  --environment-variables \
  EDGEDB_SERVER_TLS_MODE=generate_self_signed \
```

Persist the SSL certificate. We have configured EdgeDB to generate a self signed SSL certificate when it starts. However, if the container is restarted a new certificate would be generated. To preserve the certificate across failures or reboots copy the certificate files and use their contents in the `EDGEDB_SERVER_TLS_KEY` and `EDGEDB_SERVER_TLS_CERT` environment variables.

```
$ key="$( az container exec \
  --resource-group $GROUP \
  --name edgedb-container-group \
  --exec-command "cat /tmp/edgedb/edbprivkey.pem" \
  | tr -d "\r" )"
$ cert="$( az container exec \
  --resource-group $GROUP \
  --name edgedb-container-group \
  --exec-command "cat /tmp/edgedb/edbtlscert.pem" \
  | tr -d "\r" )"
$ az container delete \
  --resource-group $GROUP \
  --name edgedb-container-group \
  --yes
$ az container create \
  --resource-group $GROUP \
```

(continues on next page)

(continued from previous page)

```
--name edgedb-container-group \
--image edgedb/edgedb \
--dns-name-label edgedb \
--ports 5656 \
--secure-environment-variables \
  "EDGEDB_SERVER_PASSWORD=$PASSWORD" \
  "EDGEDB_SERVER_BACKEND_DSN=$DSN" \
  "EDGEDB_SERVER_TLS_KEY=$key" \
--environment-variables \
  "EDGEDB_SERVER_TLS_CERT=$cert"
```

To access the EdgeDB instance you've just provisioned on Azure from your local machine link the instance.

```
$ printf $PASSWORD | edgedb instance link \
  --password-from-stdin \
  --non-interactive \
  --trust-tls-cert \
  --host $( \
    az container list \
      --resource-group $GROUP \
      --query "[?name=='edgedb-container-group'].ipAddress.fqdn | [0]" \
      --output tsv ) \
  azure
```

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

You can now connect to your instance.

```
$ edgedb -I azure
```

#### 4.2.2.3 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

### 4.2.3 DigitalOcean

**edb-alt-title** Deploying EdgeDB to DigitalOcean

In this guide we show how to deploy EdgeDB to DigitalOcean either with a One-click Deploy option or a *managed PostgreSQL* database as the backend.

### 4.2.3.1 One-click Deploy

#### 4.2.3.1.1 Prerequisites

- edgedb CLI ([install](#))
- DigitalOcean account

Click the button below and follow the droplet creation workflow on DigitalOcean to deploy an EdgeDB instance.

By default, the admin password is `edgedbpassword`; let's change that to something more secure. First, find your droplet's IP address on the [DigitalOcean dashboard](#) and assign it to an environment variable `IP`.

```
$ IP=<your-droplet-ip>
```

Then use the `read` command to securely assign a value to the `PASSWORD` environment variable.

```
$ echo -n "> " && read -s PASSWORD
```

Use these variables to change the password for the default role `edgedb`.

```
$ printf edgedbpassword | edgedb query \  
  --host $IP \  
  --password-from-stdin \  
  --tls-security insecure \  
  "alter role edgedb set password := '${PASSWORD}'"  
OK: ALTER ROLE
```

## Construct the DSN

Let's construct your instance's DSN (also known as a “connection string”). We'll write the value to a file called `dsn.txt` so it doesn't get stored in shell logs.

```
$ echo edgedb://edgedb:$PASSWORD@$IP > dsn.txt
```

Copy the value from `dsn.txt`. Run the following command to open a REPL to the new instance.

```
$ edgedb --dsn <dsn> --tls-security insecure  
edgedb>
```

Success! You're now connected to your remote instance.

It's often useful to assign an alias to the remote instance using `edgedb instance link`.

```
$ edgedb instance link \  
  --dsn <dsn> \  
  --trust-tls-cert \  
  --non-interactive \  
  my_instance  
Authenticating to edgedb://edgedb@1.2.3.4:5656/edgedb  
Trusting unknown server certificate:  
SHA1:1880da9527be464e2cad3bdb20dfc430a6af5727  
Successfully linked to remote instance. To connect run:  
  edgedb -I my_instance
```

You can now use the `-I` CLI flag to execute commands against your remote instance:

```
$ edgedb -I my_instance
edgedb>
```

### 4.2.3.2 Deploy with Managed PostgreSQL

#### 4.2.3.2.1 Prerequisites

- `edgedb` CLI ([install](#))
- DigitalOcean account
- `doctl` CLI ([install](#))
- `jq` ([install](#))

#### 4.2.3.2.2 Create a managed PostgreSQL instance

If you already have a PostgreSQL instance you can skip this step.

```
$ DSN=$( \
    doctl databases create edgedb-postgres \
        --engine pg \
        --version 14 \
        --size db-s-1vcpu-1gb \
        --num-nodes 1 \
        --region sfo3 \
        --output json \
    | jq -r '.[0].connection.uri' )"
```

#### 4.2.3.2.3 Provision a droplet

Replace `SSH_KEY_IDS` with the ids for the ssh keys you want to ssh into the new droplet with. Separate multiple values with a comma. You can list your keys with `doctl compute ssh-key list`. If you don't have any ssh keys in your DigitalOcean account you can follow [this guide](#) to add one now.

```
$ IP=$( \
    doctl compute droplet create edgedb \
        --image edgedb \
        --region sfo3 \
        --size s-2vcpu-4gb \
        --ssh-keys $SSH_KEY_IDS \
        --format PublicIPv4 \
        --no-header \
        --wait )"
```

Configure the backend Postgres DSN. To simplify the initial deployment, let's instruct EdgeDB to run in insecure mode (with password authentication off and an autogenerated TLS certificate). We will secure the instance once things are up and running.

```
$ printf "EDGEDB_SERVER_BACKEND_DSN=${DSN} \
\nEDGEDB_SERVER_SECURITY=insecure_dev_mode\n" \
| ssh root@$IP -T "cat > /etc/edgedb/env"

$ ssh root@$IP "systemctl restart edgedb.service"
```

Set the superuser password.

```
$ echo -n "> " && read -s PASSWORD

$ edgedb -H $IP --tls-security insecure query \
    "alter role edgedb set password := '$PASSWORD'"
OK: ALTER ROLE
```

Set the security policy to strict.

```
$ printf "EDGEDB_SERVER_BACKEND_DSN=${DSN} \
\nEDGEDB_SERVER_SECURITY=strict\n" \
| ssh root@$IP -T "cat > /etc/edgedb/env"

$ ssh root@$IP "systemctl restart edgedb.service"
```

---

**Note:** To upgrade an existing EdgeDB droplet to the latest point release, ssh into your droplet and run the following.

```
$ apt-get update && apt-get install --only-upgrade edgedb-server-2
$ systemctl restart edgedb
```

---

That's it! Refer to the [Construct the DSN](#) section above to connect to your instance.

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

#### 4.2.3.2.4 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

### 4.2.4 Fly.io

#### edb-alt-title Deploying EdgeDB to Fly.io

In this guide we show how to deploy EdgeDB using a [Fly.io](#) PostgreSQL cluster as the backend. The deployment consists of two apps: one running Postgres and the other running EdgeDB.

---

**Note:** At the moment, it isn't possible to expose Fly-hosted EdgeDB instances to the public internet, only internally to other Fly projects. As such your application must also be hosted on Fly.

---

#### 4.2.4.1 Prerequisites

- Fly.io account
- `flyctl` CLI ([install](#))

#### 4.2.4.2 Provision a Fly.io app for EdgeDB

Every Fly.io app must have a globally unique name, including service VMs like Postgres and EdgeDB. Pick a name and assign it to a local environment variable called `EDB_APP`. In the command below, replace `myorg-edgedb` with a name of your choosing.

```
$ EDB_APP=myorg-edgedb
$ flyctl apps create --name $EDB_APP
New app created: myorg-edgedb
```

Now let's use the `read` command to securely assign a value to the `PASSWORD` environment variable.

```
$ echo -n "> " && read -s PASSWORD
```

Now let's assign this password to a Fly `secret`, plus a few other secrets that we'll need. There are a couple more environment variables we need to set:

```
$ flyctl secrets set \
  EDGEDB_PASSWORD="$PASSWORD" \
  EDGEDB_SERVER_BACKEND_DSN_ENV=DATABASE_URL \
  EDGEDB_SERVER_TLS_CERT_MODE=generate_self_signed \
  EDGEDB_SERVER_PORT=8080 \
  --app $EDB_APP
Secrets are staged for the first deployment
```

Let's discuss what's going on with all these secrets.

- The `EDGEDB_SERVER_BACKEND_DSN_ENV` tells the EdgeDB container where to look for the PostgreSQL connection string (more on that below)
- The `EDGEDB_SERVER_TLS_CERT_MODE` tells EdgeDB to auto-generate a self-signed TLS certificate. You may instead choose to provision a custom TLS certificate. In this case, you should instead create two other secrets: assign your certificate to `EDGEDB_SERVER_TLS_CERT` and your private key to `EDGEDB_SERVER_TLS_KEY`.
- Lastly, `EDGEDB_SERVER_PORT` tells EdgeDB to listen on port 8080 instead of the default 5656, because Fly.io prefers `8080` for its default health checks.

Finally, let's scale the VM as EdgeDB requires a little bit more than the default Fly.io VM side provides:

```
$ flyctl scale vm shared-cpu-1x --memory=1024 --app $EDB_APP
Scaled VM Type to
shared-cpu-1x
  CPU Cores: 1
  Memory: 1 GB
```

#### 4.2.4.3 Create a PostgreSQL cluster

Now we need to provision a PostgreSQL cluster and attach it to the EdgeDB app.

---

**Note:** If you have an existing PostgreSQL cluster in your Fly.io organization, you can skip to the attachment step.

---

Then create a new PostgreSQL cluster. This may take a few minutes to complete.

```
$ PG_APP=myorg-postgres
$ flyctl pg create --name $PG_APP --vm-size shared-cpu-1x
? Select region: sea (Seattle, Washington (US))
? Specify the initial cluster size: 1
? Volume size (GB): 10
Creating postgres cluster myorg-postgres in organization personal
Postgres cluster myorg-postgres created
  Username:    postgres
  Password:    <random password>
  Hostname:   myorg-postgres.internal
  Proxy Port: 5432
  PG Port:   5433
Save your credentials in a secure place, you won't be able to see them
again!
Monitoring Deployment
...
--> v0 deployed successfully
```

In the output, you'll notice a line that says Machine <machine-id> is created. The ID in that line is the ID of the virtual machine created for your Postgres cluster. We now need to use that ID to scale the cluster since the shared-cpu-1x VM doesn't have enough memory by default. Scale it with this command:

```
$ flyctl machine update <machine-id> --memory 512 --app $PG_APP -y
Searching for image 'flyio/postgres:14.6' remotely...
image found: img_0lq747j0ym646x35
Image: registry-1.docker.io/flyio/postgres:14.6
Image size: 361 MB

Updating machine <machine-id>
  Waiting for <machine-id> to become healthy (started, 3/3)
Machine <machine-id> updated successfully!
==> Monitoring health checks
  Waiting for <machine-id> to become healthy (started, 3/3)
...
```

With the VM scaled sufficiently, we can now attach the PostgreSQL cluster to the EdgeDB app:

```
$ PG_ROLE=myorg_edgedb
$ flyctl pg attach "$PG_APP" \
  --database-user "$PG_ROLE" \
  --app $EDB_APP
Postgres cluster myorg-postgres is now attached to myorg-edgedb
The following secret was added to myorg-edgedb:
  DATABASE_URL=postgres://...
```

Lastly, EdgeDB needs the ability to create Postgres databases and roles, so let's adjust the permissions on the role that EdgeDB will use to connect to Postgres:

```
$ echo "alter role \"\$PG_ROLE\" createrole createdb; \quit" \
    | flyctl pg connect --app \$PG_APP
...
ALTER ROLE
```

#### 4.2.4.4 Start EdgeDB

Everything is set! Time to start EdgeDB.

```
$ flyctl deploy --image=edgedb/edgedb \
    --remote-only --app \$EDB_APP
...
1 desired, 1 placed, 1 healthy, 0 unhealthy
--> v0 deployed successfully
```

That's it! You can now start using the EdgeDB instance located at `edgedb://myorg-edgedb.internal` in your Fly.io apps.

If deploy did not succeed:

1. make sure you've scaled the EdgeDB VM
2. re-run the deploy command
3. check the logs for more information: `flyctl logs --app \$EDB_APP`

#### 4.2.4.5 Persist the generated TLS certificate

Now we need to persist the auto-generated TLS certificate to make sure it survives EdgeDB app restarts. (If you've provided your own certificate, skip this step).

```
$ EDB_SECRETS="EDGEDB_SERVER_TLS_KEY EDGEDB_SERVER_TLS_CERT"
$ flyctl ssh console --app \$EDB_APP -C \
    "edgedb-show-secrets.sh --format=toml \$EDB_SECRETS" \
    | tr -d '\r' | flyctl secrets import --app \$EDB_APP
```

#### 4.2.4.6 Connecting to the instance

Let's construct the DSN (AKA “connection string”) for our instance. DSNs have the following format: `edgedb://<username>:<password>@<hostname>:<port>`. We can construct the DSN with the following components:

- <username>: the default value — edgedb
- <password>: the value we assigned to \$PASSWORD
- <hostname>: the name of your EdgeDB app (stored in the \$EDB\_APP environment variable) suffixed with .internal. Fly uses this synthetic TLD to simplify inter-app communication. Ex: myorg-edgedb.internal.
- <port>: 8080, which we configured earlier

We can construct this value and assign it to a new environment variable called DSN.

```
$ DSN=edgedb://edgedb:$PASSWORD@$EDB_APP.internal:8080
```

Consider writing it to a file to ensure the DSN looks correct. Remember to delete the file after you're done. (Printing this value to the terminal with echo is insecure and can leak your password into shell logs.)

```
$ echo $DSN > dsn.txt
$ open dsn.txt
$ rm dsn.txt
```

#### 4.2.4.6.1 From a Fly.io app

To connect to this instance from another Fly app (say, an app that runs your API server) set the value of the `EDGEDB_DSN` secret inside that app.

```
$ flyctl secrets set \
  EDGEDB_DSN=$DSN \
  --app my-other-fly-app
```

We'll also set another variable that will disable EdgeDB's TLS checks. Inter-application communication is secured by Fly so TLS isn't vital in this case; configuring TLS certificates is also beyond the scope of this guide.

```
$ flyctl secrets set EDGEDB_CLIENT_TLS_SECURITY=insecure \
  --app my-other-fly-app
```

You can also set these values as environment variables inside your `fly.toml` file, but using Fly's built-in `secrets` functionality is recommended.

#### 4.2.4.6.2 From external application

If you need to access EdgeDB from outside the Fly.io network, you'll need to configure the Fly.io proxy to let external connections in.

First, save the EdgeDB app config in an **empty directory**:

```
$ flyctl config save -a $EDB_APP
```

A `fly.toml` file will be created upon result. Let's make sure our `[[services]]` section looks something like this:

```
[[services]]
http_checks = []
internal_port = 8080
processes = ["app"]
protocol = "tcp"
script_checks = []
[services.concurrency]
hard_limit = 25
soft_limit = 20
type = "connections"

[[services.ports]]
port = 5656
```

(continues on next page)

(continued from previous page)

```
[[services.tcp_checks]]
  grace_period = "1s"
  interval = "15s"
  restart_limit = 0
  timeout = "2s"
```

In the same directory, [redeploy the EdgeDB app](#). This makes the EdgeDB port available to the outside world. You can now access the instance from any host via the following public DSN: `edgedb://edgedb:$PASSWORD@$EDB_APP.fly.dev`.

To secure communication between the server and the client, you will also need to set the `EDGEDB_TLS_CA` environment secret in your application. You can securely obtain the certificate content by running:

```
$ flyctl ssh console -a $EDB_APP \
-C "edgedb-show-secrets.sh --format=raw EDGEDB_SERVER_TLS_CERT"
```

#### 4.2.4.6.3 From your local machine

To access the EdgeDB instance from local development machine/laptop, install the Wireguard [VPN](#) and create a tunnel, as described on Fly's [Private Networking](#) docs.

Once it's up and running, use `edgedb instance link` to create a local alias to the remote instance.

```
$ edgedb instance link \
--trust-tls-cert \
--dsn $DSN \
--non-interactive \
fly
Authenticating to edgedb://edgedb@myorg-edgedb.internal:5656/edgedb
Successfully linked to remote instance. To connect run:
edgedb -I fly
```

You can now run CLI commands against this instance by specifying it by name with `-I fly`; for example, to apply migrations:

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

```
$ edgedb -I fly migrate
```

#### 4.2.4.7 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

## 4.2.5 Google Cloud

**edb-alt-title** Deploying EdgeDB to Google Cloud

In this guide we show how to deploy EdgeDB on GCP using Cloud SQL and Kubernetes.

### 4.2.5.1 Prerequisites

- Google Cloud account with billing enabled (or a [free trial](#))
- `gcloud` CLI ([install](#))
- `kubectl` CLI ([install](#))

Make sure you are logged into Google Cloud.

```
$ gcloud init
```

### 4.2.5.2 Create a project

Set the `PROJECT` environment variable to the project name you'd like to use. Google Cloud only allow letters, numbers, and hyphens.

```
$ PROJECT=edgedb
```

Then create a project with this name. Skip this step if your project already exists.

```
$ gcloud projects create $PROJECT
```

Then enable the requisite APIs.

```
$ gcloud services enable \
  container.googleapis.com \
  sqladmin.googleapis.com \
  iam.googleapis.com \
  --project=$PROJECT
```

### 4.2.5.3 Provision a Postgres instance

Use the `read` command to securely assign a value to the `PASSWORD` environment variable.

```
$ echo -n "> " && read -s PASSWORD
```

Then create a Cloud SQL instance and set the password.

```
$ gcloud sql instances create ${PROJECT}-postgres \
  --database-version=POSTGRES_13 \
  --cpu=1 \
  --memory=3840MiB \
  --region=us-west2 \
  --project=$PROJECT
$ gcloud sql users set-password postgres \
  --instance=${PROJECT}-postgres \
```

(continues on next page)

(continued from previous page)

```
--password=$PASSWORD \
--project=$PROJECT
```

#### 4.2.5.4 Create a Kubernetes cluster

Create an empty Kubernetes cluster inside your project.

```
$ gcloud container clusters create ${PROJECT}-k8s \
  --zone=us-west2-a \
  --num-nodes=1 \
  --project=$PROJECT
```

#### 4.2.5.5 Configure service account

Create a new service account, configure its permissions, and generate a `credentials.json` file.

```
$ gcloud iam service-accounts create ${PROJECT}-account \
  --project=$PROJECT

$ MEMBER="${PROJECT}-account@${PROJECT}.iam.gserviceaccount.com"
$ gcloud projects add-iam-policy-binding $PROJECT \
  --member=serviceAccount:${MEMBER} \
  --role=roles/cloudsql.admin \
  --project=$PROJECT

$ gcloud iam service-accounts keys create credentials.json \
  --iam-account=${MEMBER}
```

Then use this `credentials.json` to authenticate the Kubernetes CLI tool `kubectl`.

```
$ kubectl create secret generic cloudsqli-instance-credentials \
  --from-file=credentials.json=credentials.json

$ INSTANCE_CONNECTION_NAME=$( \
  gcloud sql instances describe ${PROJECT}-postgres \
  --format="value(connectionName)" \
  --project=$PROJECT
)

$ DSN="postgresql://postgres:${PASSWORD}@127.0.0.1:5432"
$ kubectl create secret generic cloudsqli-db-credentials \
  --from-literal=dsn=$DSN \
  --from-literal=password=$PASSWORD \
  --from-literal=instance=${INSTANCE_CONNECTION_NAME}=tcp:5432
```

#### 4.2.5.6 Deploy EdgeDB

Download the starter EdgeDB Kubernetes configuration file. This file specifies a persistent volume, a container running a Cloud SQL authorization proxy, and a container to run EdgeDB itself. It relies on the secrets we declared in the previous step.

```
$ wget "https://raw.githubusercontent.com\\
/edgedb/edgedb-deploy/dev/gcp/deployment.yaml"

$ kubectl apply -f deployment.yaml
```

Ensure the pods are running.

```
$ kubectl get pods
NAME           READY   STATUS        RESTARTS   AGE
edgedb-977b8fdf6-jswlw   0/2     ContainerCreating   0          16s
```

The READY 0/2 tells us neither of the two pods have finished booting. Re-run the command until 2/2 pods are READY.

If there were errors you can check EdgeDB's logs with:

```
$ kubectl logs deployment/edgedb --container edgedb
```

#### 4.2.5.7 Persist TLS Certificate

Now that our EdgeDB instance is up and running, we need to download a local copy of its self-signed TLS certificate (which it generated on startup) and pass it as a secret into Kubernetes. Then we'll redeploy the pods.

```
$ kubectl create secret generic cloudsqtlstls-credentials \
--from-literal=tlskey="$(
    kubectl exec deploy/edgedb -- \
        edgedb-show-secrets.sh --format=raw EDGEDB_SERVER_TLS_KEY
)" \
--from-literal=tlscert="$((
    kubectl exec deploy/edgedb -- \
        edgedb-show-secrets.sh --format=raw EDGEDB_SERVER_TLS_CERT
))"

$ kubectl delete -f deployment.yaml

$ kubectl apply -f deployment.yaml
```

#### 4.2.5.8 Expose EdgeDB

```
$ kubectl expose deploy/edgedb --type LoadBalancer
```

#### 4.2.5.9 Get your instance's DSN

Get the public-facing IP address of your database.

```
$ kubectl get service
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)
edgedb   LoadBalancer   <ip>        <ip>        5656:30841/TCP
```

Copy and paste the EXTERNAL-IP associated with the service named edgedb. With this IP address, you can construct your instance's [DSN](#):

```
$ EDGEDEB_IP=<copy IP address here>
$ EDGEDEB_DSN="edgedb://edgedb:${PASSWORD}@${EDGEDEB_IP}"
```

To print the final DSN, you can echo it. Note that you should only run this command on a computer you trust, like a personal laptop or sandboxed environment.

```
$ echo $EDGEDEB_DSN
```

The resulting DSN can be used to connect to your instance. To test it, try opening a REPL:

```
$ edgedb --dsn $EDGEDEB_DSN --tls-security insecure
EdgeDB 2.x (repl 2.x)
Type \help for help, \quit to quit.
edgedb> select "hello world!";
```

#### 4.2.5.9.1 In development

To make this instance easier to work with during local development, create an alias using `edgedb instance link`.

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

```
$ echo $PASSWORD | edgedb instance link \
--dsn $EDGEDEB_DSN \
--password-from-stdin \
--non-interactive \
--trust-tls-cert \
gcp_instance
```

You can now refer to the remote instance using the alias instance on your machine called `gcp_instance`. You can use this alias wherever an instance name is expected; for instance, you can open a REPL:

```
$ edgedb -I gcp_instance
```

Or apply migrations:

```
$ edgedb -I gcp_instance migrate
```

### 4.2.5.9.2 In production

To connect to this instance in production, set the `EDGEDB_DSN` environment variable wherever you deploy your application server; EdgeDB's client libraries read the value of this variable to know how to connect to your instance.

### 4.2.5.10 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

## 4.2.6 Heroku

### edb-alt-title Deploying EdgeDB to Heroku

**Warning:** Deployment to Heroku is currently not working due to a change in Heroku's Postgres extension schema. We plan to implement changes to address this and will update this guide to remove this warning once we have done so. See [the relevant Github issue](#) for more information or to subscribe to notifications.

In this guide we show how to deploy EdgeDB to Heroku using a Heroku PostgreSQL add-on as the backend.

Because of Heroku's architecture EdgeDB must be deployed with a web app on Heroku. For this guide we will use a todo app written in Node.

### 4.2.6.1 Prerequisites

- Heroku account
- heroku CLI ([install](#))
- Node.js ([install](#))

### 4.2.6.2 Setup

First copy the code, initialize a new git repo, and create a new heroku app.

```
$ npx degit 'edgedb/simpletodo#main' simpletodo-heroku
$ cd simpletodo-heroku
$ git init --initial-branch main
$ heroku apps:create --buildpack heroku/nodejs
$ edgedb project init --non-interactive
```

If you are using the [JS query builder for EdgeDB](#) then you will need to check the `dbschema/edgeql-js` directory in to your git repo after running `yarn edgeql-js`. The `edgeql-js` command cannot be run during the build step on Heroku because it needs access to a running EdgeDB instance which is not available at build time on Heroku.

```
$ yarn install && npx @edgedb/generate edgeql-js
```

The `dbschema/edgeql-js` directory was added to the `.gitignore` in the upstream project so we'll remove it here.

```
$ sed -i '/^dbschema\/edgeql-js$/d' .gitignore
```

#### 4.2.6.3 Create a PostgreSQL Add-on

Heroku's smallest PostgreSQL plan, Hobby Dev, limits the number of rows to 10,000, but EdgeDB's standard library uses more than 20,000 rows so we need to use a different plan. We'll use the [Standard 0 plan](#) for this guide.

```
$ heroku addons:create heroku-postgresql:standard-0
```

#### 4.2.6.4 Add the EdgeDB Buildpack

To run EdgeDB on Heroku we'll add the [EdgeDB buildpack](#).

```
$ heroku buildpacks:add \
  --index 1 \
  https://github.com/edgedb/heroku-buildpack-edgedb.git
```

#### 4.2.6.5 Use start-edgedb in the Procfile

To make EdgeDB available to a process prepend the command with `start-edgedb` which is provided by the EdgeDB buildpack. For the sample application in this guide, the web process is started with the command `npm start`. If you have other processes in your application besides/instead of web that need to access EdgeDB those process commands should be prepended with `start-edgedb` too.

```
$ echo "web: start-edgedb npm start" > Procfile
```

#### 4.2.6.6 Deploy the App

Commit the changes and push to Heroku to deploy the app.

```
$ git add .
$ git commit -m "first commit"
$ git push heroku main
```

#### 4.2.6.7 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

### 4.2.7 Docker

**edb-alt-title** Deploying EdgeDB with Docker

#### 4.2.7.1 When to use the edgedb/edgedb Docker image

This image is primarily intended to be used directly when there is a requirement to use Docker containers, such as in production, or in a development setup that involves multiple containers orchestrated by Docker Compose or a similar tool. Otherwise, using the [edgedb server](#) CLI on the host system is the recommended way to install and run EdgeDB servers.

#### 4.2.7.2 How to use this image

The simplest way to run the image (without data persistence) is this:

```
$ docker run --name edgedb -d \
-e EDGEDB_SERVER_SECURITY=insecure_dev_mode \
edgedb/edgedb
```

See the [Configuration](#) section below for the meaning of the EDGEDB\_SERVER\_SECURITY variable and other options.

Then, to authenticate to the EdgeDB instance and store the credentials in a Docker volume, run:

```
$ docker run -it --rm --link=edgedb \
-e EDGEDB_SERVER_PASSWORD=secret \
-v edgedb-cli-config:/config/edgedb edgedb/edgedb-cli \
-H edgedb instance link my_instance
```

Now, to open an interactive shell to the database instance run this:

```
$ docker run -it --rm --link=edgedb \
-v edgedb-cli-config:/config/edgedb edgedb/edgedb-cli \
-I my_instance
```

#### 4.2.7.3 Data Persistence

If you want the contents of the database to survive container restarts, you must mount a persistent volume at the path specified by EDGEDB\_SERVER\_DATADIR (`/var/lib/edgedb/data` by default). For example:

```
$ docker run \
--name edgedb \
-e EDGEDB_SERVER_PASSWORD=secret \
-e EDGEDB_SERVER_TLS_MODE=generate_self_signed \
-v /my/data/directory:/var/lib/edgedb/data \
-d edgedb/edgedb
```

Note that on Windows you must use a Docker volume instead:

```
$ docker volume create --name=edgedb-data
$ docker run \
--name edgedb \
-e EDGEDB_SERVER_PASSWORD=secret \
-e EDGEDB_SERVER_TLS_MODE=generate_self_signed \
-v edgedb-data:/var/lib/edgedb/data \
-d edgedb/edgedb
```

It is also possible to run an `edgedb` container on a remote PostgreSQL cluster specified by `EDGEDB_SERVER_BACKEND_DSN`. See below for details.

#### 4.2.7.4 Schema Migrations

A derived image may include application schema and migrations in `/dbschema`, in which case the container will attempt to apply the schema migrations found in `/dbschema/migrations`, unless the `EDGEDB_DOCKER_APPLY.Migrations` environment variable is set to `never`.

#### 4.2.7.5 Docker Compose

A simple `docker-compose` configuration might look like this. With a `docker-compose.yaml` containing:

```
version: "3"
services:
  edgedb:
    image: edgedb/edgedb
    environment:
      EDGEDB_SERVER_SECURITY: insecure_dev_mode
    volumes:
      - "./dbschema:/dbschema"
    ports:
      - "5656:5656"
```

Once there is a `schema` in `dbschema/` a migration can be created with:

```
$ edgedb --tls-security=insecure -P 5656 migration create
```

Alternatively, if you don't have the EdgeDB CLI installed on your host machine, you can use the CLI bundled with the server container:

```
$ docker-compose exec edgedb edgedb --tls-security=insecure migration create
```

#### 4.2.7.6 Configuration

The Docker image supports the same set of environment variables as the EdgeDB server process, which are documented under [Reference > Environment Variables](#).

EdgeDB containers can be additionally configured using initialization scripts and some Docker-specific environment variables, documented below.

---

**Note:** Some variables support `_ENV` and `_FILE` variants to support more advanced configurations.

---

##### 4.2.7.6.1 Initial configuration

When an EdgeDB container starts on the specified data directory or remote Postgres cluster for the first time, initial instance setup is performed. This is called the *bootstrap phase*.

The following environment variables affect the bootstrap only and have no effect on subsequent container runs.

## **EDGEDB\_SERVER\_BOOTSTRAP\_COMMAND**

Useful to fine-tune initial user and database creation, and other initial setup. If neither the `EDGEDB_SERVER_BOOTSTRAP_COMMAND` variable or the `EDGEDB_SERVER_BOOTSTRAP_SCRIPT_FILE` are explicitly specified, the container will look for the presence of `/edgedb-bootstrap.edgeql` in the container (which can be placed in a derived image).

Maps directly to the `edgedb-server` flag `--default-auth-method`. The `*_FILE` and `*_ENV` variants are also supported.

## **EDGEDB\_SERVER\_BOOTSTRAP\_SCRIPT\_FILE**

Deprecated in image version 2.8: use `EDGEDB_SERVER_BOOTSTRAP_COMMAND_FILE` instead.

Run the script when initializing the database. The script is run by default user within default database.

## **EDGEDB\_SERVER\_PASSWORD**

The password for the default superuser account will be set to this value. If no value is provided a password will not be set, unless set via `EDGEDB_SERVER_BOOTSTRAP_COMMAND`. (If a value for `EDGEDB_SERVER_BOOTSTRAP_COMMAND` is provided, this variable will be ignored.)

The `*_FILE` and `*_ENV` variants are also supported.

## **EDGEDB\_SERVER\_PASSWORD\_HASH**

A variant of `EDGEDB_SERVER_PASSWORD`, where the specified value is a hashed password verifier instead of plain text.

If `EDGEDB_SERVER_BOOTSTRAP_COMMAND` is set, this variable will be ignored.

The `*_FILE` and `*_ENV` variants are also supported.

## **EDGEDB\_SERVER\_GENERATE\_SELF\_SIGNED\_CERT**

**Warning:** Deprecated: use `EDGEDB_SERVER_TLS_CERT_MODE=generate_self_signed` instead.

Set this option to 1 to tell the server to automatically generate a self-signed certificate with key file in the `EDGEDB_SERVER_DATADIR` (if present, see below), and echo the certificate content in the logs. If the certificate file exists, the server will use it instead of generating a new one.

Self-signed certificates are usually used in development and testing, you should likely provide your own certificate and key file with the variables below.

## **EDGEDB\_SERVER\_TLS\_CERT/EDGEDB\_SERVER\_TLS\_KEY**

The TLS certificate and private key data, exclusive with `EDGEDB_SERVER_TLS_MODE=generate_self_signed`. The `*_FILE` and `*_ENV` variants are also supported.

### **Custom scripts in `/docker-entrypoint.d/`**

To perform additional initialization, a derived image may include one or more executable files in `/docker-entrypoint.d/`, which will get executed by the container entrypoint *before* any other processing takes place.

#### **4.2.7.6.2 Runtime configuration**

## **EDGEDB\_DOCKER\_LOG\_LEVEL**

Determines the log verbosity level in the entrypoint script. Valid levels are `trace`, `debug`, `info`, `warning`, and `error`. The default is `info`.

### **Custom scripts in `/edgedb-bootstrap.d/` and `/edgedb-bootstrap-late.d`**

To perform additional initialization, a derived image may include one or more `*.edgeql` or `*.sh` scripts, which are executed in addition to and *after* the initialization specified by the environment variables above or the `/edgedb-bootstrap.edgeql` script. Parts in `/edgedb-bootstrap.d` are executed *before* any schema migrations are applied, and parts in `/edgedb-bootstrap-late.d` are executed *after* the schema migration have been applied.

#### **4.2.7.7 Health Checks**

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

## **4.2.8 Bare Metal**

### **edb-alt-title Deploying EdgeDB to a Bare Metal Server**

In this guide we show how to deploy EdgeDB to bare metal using your system's package manager and `systemd`.

#### **4.2.8.1 Install the EdgeDB Package**

The steps for installing the EdgeDB package will be slightly different depending on your Linux distribution. Once you have the package installed you can jump to [Enable a systemd unit](#).

### 4.2.8.1.1 Debian/Ubuntu LTS

Import the EdgeDB packaging key.

```
$ sudo mkdir -p /usr/local/share/keyrings && \
  sudo curl --proto '=https' --tlsv1.2 -sSf \
  -o /usr/local/share/keyrings/edgedb-keyring.gpg \
  https://packages.edgedb.com/keys/edgedb-keyring.gpg
```

Add the EdgeDB package repository.

```
$ echo deb [signed-by=/usr/local/share/keyrings/edgedb-keyring.gpg] \
  https://packages.edgedb.com/apt \
  $(grep "VERSION_CODENAME=" /etc/os-release | cut -d= -f2) main \
  | sudo tee /etc/apt/sources.list.d/edgedb.list
```

Install the EdgeDB package.

```
$ sudo apt-get update && sudo apt-get install edgedb-2
```

### 4.2.8.1.2 CentOS/RHEL 7/8

Add the EdgeDB package repository.

```
$ sudo curl --proto '=https' --tlsv1.2 -sSfL \
  https://packages.edgedb.com/rpm/edgedb-rhel.repo \
  > /etc/yum.repos.d/edgedb.repo
```

Install the EdgeDB package.

```
$ sudo yum install edgedb-2
```

### 4.2.8.2 Enable a systemd unit

The EdgeDB package comes bundled with a systemd unit that is disabled by default. You can start the server by enabling the unit.

```
$ sudo systemctl enable --now edgedb-server-2
```

This will start the server on port 5656, and the data directory will be `/var/lib/edgedb/1/data`.

**Warning:** `edgedb-server` cannot be run as root.

#### 4.2.8.3 Set environment variables

To set environment variables when running EdgeDB with `systemctl`,

```
$ systemctl edit --full edgedb-server-2
```

This opens a `systemd` unit file. Set the desired environment variables under the `[Service]` section. View the supported environment variables at [Reference > Environment Variables](#).

##### `[Service]`

```
Environment="EDGEDB_SERVER_TLS_CERT_MODE=generate_self_signed"
Environment="EDGEDB_SERVER_ADMIN_UI=enabled"
```

Save the file and exit, then restart the service.

```
$ systemctl restart edgedb-server-2
```

#### 4.2.8.4 Set a password

There is no default password. Set a password by connecting from localhost.

```
$ echo -n "> " && read -s PASSWORD
$ sudo edgedb --port 5656 --tls-security insecure --admin query \
  "ALTER ROLE edgedb SET password := '$PASSWORD'"
```

The server listens on localhost by default. Changing this looks like this.

```
$ edgedb --port 5656 --tls-security insecure --password query \
  "CONFIGURE INSTANCE SET listen_addresses := {'0.0.0.0'};"
```

The listen port can be changed from the default 5656 if your deployment scenario requires a different value.

```
$ edgedb --port 5656 --tls-security insecure --password query \
  "CONFIGURE INSTANCE SET listen_port := 1234;"
```

You may need to restart the server after changing the listen port or addresses.

```
$ sudo systemctl restart edgedb-server-2
```

#### 4.2.8.5 Link the instance with the CLI

The following is an example of linking a bare metal instance that is running on localhost. This command assigns a name to the instance, to make it more convenient to refer to when running CLI commands.

```
$ edgedb instance link \
  --host localhost \
  --port 5656 \
  --user edgedb \
  --database edgedb \
  --trust-tls-cert \
  bare_metal_instance
```

This allows connecting to the instance with its name.

```
$ edgedb -I bare_metal_instance
```

#### 4.2.8.6 Upgrading EdgeDB

---

**Note:** The command groups `edgedb instance` and `edgedb project` are not intended to manage production instances.

---

When you want to upgrade to the newest point release upgrade the package and restart the `edgedb-server-2` unit.

##### 4.2.8.6.1 Debian/Ubuntu LTS

```
$ sudo apt-get update && sudo apt-get install --only-upgrade edgedb-2  
$ sudo systemctl restart edgedb-server-2
```

##### 4.2.8.6.2 CentOS/RHEL 7/8

```
$ sudo yum update edgedb-2  
$ sudo systemctl restart edgedb-server-2
```

#### 4.2.8.7 Health Checks

Using an HTTP client, you can perform health checks to monitor the status of your EdgeDB instance. Learn how to use them with our [health checks guide](#).

#### 4.2.9 Health Checks

You may want to monitor the status of your EdgeDB instance. Is it up? Is it ready to take queries? This guide will show you to perform health checks using HTTP and the `alive` and `ready` endpoints.

##### 4.2.9.1 Check Instance Aliveness

To check if the instance is alive, make a request to this endpoint:

```
http://<hostname>:<port>/server/status/alive
```

To find your `<port>`, you can run `edgedb instance list` to see a table of all your instances along with their port numbers.

The endpoint will respond with a `200` status code and "OK" as the payload if the server is alive. If not, you will receive a `50x` code or a network error.

#### 4.2.9.2 Check Instance Readiness

To check if the instance is ready, make a request to this endpoint:

```
http://<hostname>:<port>/server/status/ready
```

As with the `alive` endpoint, you can find your `<port>` by running `edgedb instance list` to see a table of all your instances along with their port numbers.

The endpoint will respond with a `200` status code and "OK" as the payload if the server is ready. If not, you will receive a `50x` code or a network error.

## 4.3 Migration Patterns

### 4.3.1 Making a property required

This example shows how a property may evolve to be more and more strict over time by looking at a user name field. However, similar evolution may be applicable to other properties that start off with few restrictions and gradually become more constrained and formalized as the needs of the project evolve.

We'll start with a fairly simple schema:

```
type User {
    property name -> str;
}
```

```
type User {
    name: str;
}
```

At this stage we don't think that this property needs to be unique or even required. Perhaps it's only used as a screen name and not as a way of identifying users.

```
$ edgedb migration create
did you create object type 'default::User'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00001.edgeql, id:
m14gwyorqqipfg7riexvbdq5dhgv7x6buqw2jaaulilcmywinmakzq
$ edgedb migrate
Applied m14gwyorqqipfg7riexvbdq5dhgv7x6buqw2jaaulilcmywinmakzq
(00001.edgeql)
```

We've got our first migration to set up the schema. Now after using that for a little while we realize that we want to make `name` a *required property*. So we make the following change in the schema file:

```
type User {
    required property name -> str;
}
```

```
type User {
    required name: str;
}
```

Next we try to migrate:

```
$ edgedb migration create
did you make property 'name' of object type 'default::User' required?
[y,n,l,c,b,s,q,?]
> y
Please specify an expression to populate existing objects in order to make
property 'name' of object type 'default::User' required:
fill_expr> 'change me'
```

Oh! That's right, we can't just make `name` *required* because there could be existing `User` objects without a `name` at all. So we need to provide some kind of placeholder value for those cases. We type '`change me`' (although any other string would do, too). This is different from specifying a `default` value since it will be applied to *existing* objects, whereas the `default` applies to *new ones*. We then run `edgedb migrate` to apply the changes.

Next we realize that we actually want to make names unique, perhaps to avoid confusion or to use them as reliable human-readable identifiers (unlike `id`). We update the schema again:

```
type User {
    required property name -> str {
        constraint exclusive;
    }
}
```

```
type User {
    required name: str {
        constraint exclusive;
    }
}
```

Now we proceed with the migration:

```
$ edgedb migration create
did you create constraint 'std::exclusive' of property 'name'?
[y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00003.edgeql, id:
m1dxs3xbk4f3vhmqh6mjzetojafddtwlphp5a3kfbfuyvupjafevy
$ edgedb migrate
edgedb error: ConstraintViolationError: name violates exclusivity
constraint
```

Some objects must have the same `name`, so the migration can't be applied. We have a couple of options for fixing this:

- 1) Review the existing data and manually `update` the entries with duplicate names so that they are unique.
- 2) Edit the migration to add an `update` which will de-duplicate `name` for any potential existing `User` objects.

The first option is good for situations where we want to signal to any other maintainer of a copy of this project that they need to make a decision about handling name duplicates in whatever way is appropriate to them without making an implicit decision once and for all.

Here we will go with the second option, which is good for situations where we know enough about the situation that we can make a decision now and never have to duplicate this effort for any other potential copies of our project.

We edit the last migration file `00003.edgeql`:

```

CREATE MIGRATION m1dxs3xbk4f3vhmqh6mjzetojafddtwlphp5a3kfbfuyvupjafevy
    ONTO m1ndhbxx7yudb2dv7zypy12su2oygyj1ggk3olryb5uszofrfml4uq
{
+   with U := default::User
+   update default::User
+   filter U.name = .name and U != default::User
+   set {
+       # De-duplicate names by appending a random uuid.
+       name := .name ++ '_' ++ <str>uuid_generate_v1mc()
+   };
+
    ALTER TYPE default::User {
        ALTER PROPERTY name {
            CREATE CONSTRAINT std::exclusive;
        };
    };
}
;
```

And then we apply the migration:

```

$ edgedb migrate
edgedb error: could not read migrations in ./dbschema/migrations: could not
read migration file ./dbschema/migrations/00003.edgeql: migration name
should be `m1t6slgcfne35vir2lcgnqkmaxsxlzvn2hanr6mijbj5esefsp7za` but `
m1dxs3xbk4f3vhmqh6mjzetojafddtwlphp5a3kfbfuyvupjafevy` is used instead.
Migration names are computed from the hash of the migration contents. To
proceed you must fix the statement to read as:
    CREATE MIGRATION m1t6slgcfne35vir2lcgnqkmaxsxlzvn2hanr6mijbj5esefsp7za
    ONTO ...
if this migration is not applied to any database. Alternatively, revert the
changes to the file.
```

The migration tool detected that we've altered the file and asks us to update the migration name (acting as a checksum) if this was deliberate. This is done as a precaution against accidental changes. Since we've done this on purpose, we can update the file and run `edgedb migrate` again.

Finally, we evolved our schema all the way from having an optional property `name` all the way to making it both *required* and *exclusive*. We've worked with the EdgeDB [migration tools](#) to iron out the kinks throughout the migration process. At this point we take a quick look at the way duplicate User objects were resolved to decide whether we need to do anything more. We can use `re_test()` to find names that look like they are ending in a UUID:

```

db> select User { name }
... filter
...     re_test('.* [a-zA-Z0-9]{8}(-[a-zA-Z0-9]{4}){3}-[a-zA-Z0-9]{12}$', .name);
{
    default::User {name: 'change me bc30d45a-2bcf-11ec-a6c2-6ff21f33a302'},
    default::User {name: 'change me bc30d8a6-2bcf-11ec-a6c2-4f739d559598'},
}
```

Looks like the only duplicates are the users that had no names originally and that never updated the 'change me' placeholders, so we can probably let them be for now. In hindsight, it may have been a good idea to use UUID-based names to populate the empty properties from the very beginning.

### 4.3.2 Adding backlinks

This example shows how to handle a schema that makes use of a backlink. We'll use a linked-list structure to represent a sequence of events.

We'll start with this schema:

```
type Event {
    required property name -> str;
    link prev -> Event;

    # ... more properties and links
}
```

```
type Event {
    required name: str;
    prev: Event;

    # ... more properties and links
}
```

We specify a `prev` link because that will make adding a new `Event` at the end of the chain easier, since we'll be able to specify the payload and the chain the `Event` should be appended to in a single `insert`. Once we've updated the schema file we proceed with our first migration:

```
$ edgedb migration create
did you create object type 'default::Event'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00001.edgeql, id:
m1v3ahcx5f43y6mlsdmlz2agnf6msbc7rt3zstiqmezaqx4ev2qovq
$ edgedb migrate
Applied m1v3ahcx5f43y6mlsdmlz2agnf6msbc7rt3zstiqmezaqx4ev2qovq
(00001.edgeql)
```

We now have a way of chaining events together. We might create a few events like these:

```
db> select Event {
...     name,
...     prev: { name },
... };
{
    default::Event {name: 'setup', prev: {}},
    default::Event {name: 'work', prev: default::Event {name: 'setup'}},
    default::Event {name: 'cleanup', prev: default::Event {name: 'work'}},
}
```

It seems like having a `next` link would be useful, too. So we can define it as a computed link by using `backlink` notation:

```
type Event {
    required property name -> str;

    link prev -> Event;
    link next := .<prev[is Event];
}
```

```
type Event {
    required name: str;

    prev: Event;
    link next := .<prev[is Event];
}
```

The migration is straightforward enough:

```
$ edgedb migration create
did you create link 'next' of object type 'default::Event'?
[y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00002.edgeql, id:
m1qpukyvw2m4lmomoseni7vdmevk4wzgsbviojacyrqgiyqjp5sd़sa
$ edgedb migrate
Applied m1qpukyvw2m4lmomoseni7vdmevk4wzgsbviojacyrqgiyqjp5sd़sa
(00002.edgeql)
```

Trying out the new link on our existing data gives us:

```
db> select Event {
...     name,
...     prev_name := .prev.name,
...     next_name := .next.name,
... };
{
    default::Event {
        name: 'setup',
        prev_name: {},
        next_name: {'work'},
    },
    default::Event {
        name: 'work',
        prev_name: 'setup',
        next_name: {'cleanup'},
    },
    default::Event {
        name: 'cleanup',
        prev_name: 'work',
        next_name: {},
    },
}
```

That's not quite right. The value of `next_name` appears to be a set rather than a singleton. This is because the link `prev` is many-to-one and so `next` is one-to-many, making it a *multi* link. Let's fix that by making the link `prev` a one-to-one, after all we're interested in building event chains, not trees.

```
type Event {
    required property name -> str;

    link prev -> Event {
        constraint exclusive;
```

(continues on next page)

(continued from previous page)

```
};

link next := .<prev[is Event];
}
```

```
type Event {
    required name: str;

    prev: Event {
        constraint exclusive;
    };
    link next := .<prev[is Event];
}
```

Since the `next` link is computed, the migration should not need any additional user input even though we're reducing the link's cardinality:

```
$ edgedb migration create
did you create constraint 'std::exclusive' of link 'prev'?
[y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00003.edgeql, id:
m17or2bfywuckdqeornjmjh7c2voxgatspcewayefcd4p2vbdepimoa
$ edgedb migrate
Applied m17or2bfywuckdqeornjmjh7c2voxgatspcewayefcd4p2vbdepimoa
(00003.edgeql)
```

The new `next` computed link is now inferred as a single link and so the query results for `next_name` and `prev_name` are symmetrical:

```
db> select Event {
...     name,
...     prev_name := .prev.name,
...     next_name := .next.name,
... };
{
    default::Event {name: 'setup', prev_name: {}, next_name: 'work'},
    default::Event {name: 'work', prev_name: 'setup', next_name: 'cleanup'},
    default::Event {name: 'cleanup', prev_name: 'work', next_name: {}},
}
```

### 4.3.3 Changing the type of a property

This example shows how to change the type of a property. We'll use a character in an adventure game as the type of data we will evolve.

Let's start with this schema:

```
type Character {
    required property name -> str;
    required property description -> str;
}
```

```
type Character {
    required name: str;
    required description: str;
}
```

We edit the schema file and perform our first migration:

```
$ edgedb migration create
did you create object type 'default::Character'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00001.edgeql, id:
m1paw3ogpsdtxaoywd6pl6beg2g64zj4ykh43zby4eqh64yjad47a
$ edgedb migrate
Applied m1paw3ogpsdtxaoywd6pl6beg2g64zj4ykh43zby4eqh64yjad47a
(00001.edgeql)
```

The intent is for the `description` to provide some text which serves both as something to be shown to the player as well as determining some game actions. So we end up with something like this:

```
db> select Character {name, description};
{
    default::Character {name: 'Alice', description: 'Tall and strong'},
    default::Character {name: 'Billie', description: 'Smart and aloof'},
    default::Character {name: 'Cameron', description: 'Dashing and smooth'},
}
```

However, as we keep developing our game it becomes apparent that this is less of a “description” and more of a “character class”, so at first we just rename the property to reflect that:

```
type Character {
    required property name -> str;
    required property class -> str;
}
```

```
type Character {
    required name: str;
    required class: str;
}
```

The migration gives us this:

```
$ edgedb migration create
did you rename property 'description' of object type 'default::Character'
to 'class'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00002.edgeql, id:
m1ljrgrfsqkvo5hsxc62mnztdhlerxp6ucdto262se6dinhuj4mqq
$ edgedb migrate
Applied m1ljrgrfsqkvo5hsxc62mnztdhlerxp6ucdto262se6dinhuj4mqq
(00002.edgeql)
```

EdgeDB detected that the change looked like a property was being renamed, which we confirmed. Since this was an existing property being renamed, the data is all preserved:

```
db> select Character {name, class};
{
    default::Character {name: 'Alice', class: 'Tall and strong'},
    default::Character {name: 'Billie', class: 'Smart and aloof'},
    default::Character {name: 'Cameron', class: 'Dashing and smooth'},
}
```

The contents of the `class` property are a bit too verbose, so we decide to update them. In order for this update to be consistently applied across several developers, we will make it in the form of a *data migration*:

```
$ edgedb migration create --allow-empty
Created ./dbschema/migrations/00003.edgeql, id:
m1qv2pdksjxxzlnujfed4b6to2ppuodj3xqax4p3r75yfef7kd7jna
```

Now we can edit the file `00003.edgeql` directly:

```
CREATE MIGRATION m1qv2pdksjxxzlnujfed4b6to2ppuodj3xqax4p3r75yfef7kd7jna
    ONTO m1ljrgrofsqkvo5hsxc62mnztdhlerxp6ucdto262se6dinhuj4mqq
{
+    update default::Character
+    set {
+        class :=
+            'warrior' if .class = 'Tall and strong' else
+            'scholar' if .class = 'Smart and aloof' else
+            'rogue'
+    };
};
```

We're ready to apply the migration:

```
$ edgedb migrate
edgedb error: could not read migrations in ./dbschema/migrations:
could not read migration file ./dbschema/migrations/00003.edgeql:
migration name should be
`m1ryafvp24g5eqjeu65zr4bqf6m3qath3lckfdhoecfnncmr7zshehq`
but `m1qv2pdksjxxzlnujfed4b6to2ppuodj3xqax4p3r75yfef7kd7jna` is used
instead.
Migration names are computed from the hash of the migration
contents. To proceed you must fix the statement to read as:
CREATE MIGRATION m1ryafvp24g5eqjeu65zr4bqf6m3qath3lckfdhoecfnncmr7zshehq
    ONTO ...
if this migration is not applied to any database. Alternatively,
revert the changes to the file.
```

The migration tool detected that we've altered the file and asks us to update the migration name (acting as a checksum) if this was deliberate. This is done as a precaution against accidental changes. Since we've done this on purpose, we can update the file and run `edgedb migrate` again.

As the game becomes more stable there's no reason for the `class` to be a `str` anymore, instead we can use an `enum` to make sure that we don't accidentally use some invalid value for it.

```
scalar type CharacterClass extending enum<warrior, scholar, rogue>;
```

```
type Character {
```

(continues on next page)

(continued from previous page)

```
required property name -> str;
required property class -> CharacterClass;
}
```

```
scalar type CharacterClass extending enum<warrior, scholar, rogue>;

type Character {
    required name: str;
    required class: CharacterClass;
}
```

Fortunately, we've already updated the class strings to match the `enum` values, so that a simple cast will convert all the values. If we had not done this earlier we would need to do it now in order for the type change to work.

```
$ edgedb migration create
did you create scalar type 'default::CharacterClass'? [y,n,l,c,b,s,q,?]
> y
did you alter the type of property 'class' of object type
'default::Character'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00004.edgeql, id:
m1hc4ynnkejef2hh7fvymvg3f26nmynpffksg7yvfksquif6lulgq
$ edgedb migrate
Applied m1hc4ynnkejef2hh7fvymvg3f26nmynpffksg7yvfksquif6lulgq
(00004.edgeql)
```

The final migration converted all the `class` property values:

```
db> select Character {name, class};
{
    default::Character {name: 'Alice', class: warrior},
    default::Character {name: 'Billie', class: scholar},
    default::Character {name: 'Cameron', class: rogue},
}
```

#### 4.3.4 Changing a property to a link

This example shows how to change a property into a link. We'll use a character in an adventure game as the type of data we will evolve.

Let's start with this schema:

```
scalar type CharacterClass extending enum<warrior, scholar, rogue>;

type Character {
    required property name -> str;
    required property class -> CharacterClass;
}
```

```
scalar type CharacterClass extending enum<warrior, scholar, rogue>;
```

(continues on next page)

(continued from previous page)

```
type Character {
    required name: str;
    required class: CharacterClass;
}
```

We edit the schema file and perform our first migration:

```
$ edgedb migration create
did you create scalar type 'default::CharacterClass'? [y,n,l,c,b,s,q,?]
> y
did you create object type 'default::Character'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00001.edgeql, id:
m1fg76t7fbvguwhkmzrx7jwki6jxr6dvkszeepd5v66oxg27ymkcq
$ edgedb migrate
Applied m1fg76t7fbvguwhkmzrx7jwki6jxr6dvkszeepd5v66oxg27ymkcq
(00001.edgeql)
```

The initial setup may look something like this:

```
db> select Character {name, class};
{
    default::Character {name: 'Alice', class: warrior},
    default::Character {name: 'Billie', class: scholar},
    default::Character {name: 'Cameron', class: rogue},
}
```

After some development work we decide to add more details about the available classes and encapsulate that information into its own type. This way instead of a property `class` we want to end up with a link `class` to the new data structure. Since we cannot just `cast` a scalar into an object, we'll need to convert between the two explicitly. This means that we will need to have both the old and the new “class” information to begin with:

```
scalar type CharacterClass extending enum<warrior, scholar, rogue>;

type NewClass {
    required property name -> str;
    multi property skills -> str;
}

type Character {
    required property name -> str;
    required property class -> CharacterClass;
    link new_class -> NewClass;
}
```

```
scalar type CharacterClass extending enum<warrior, scholar, rogue>;

type NewClass {
    required name: str;
    multi skills: str;
}
```

(continues on next page)

(continued from previous page)

```
type Character {
    required name: str;
    required class: CharacterClass;
    new_class: NewClass;
}
```

We update the schema file and migrate to the new state:

```
$ edgedb migration create
did you create object type 'default::NewClass'? [y,n,l,c,b,s,q,?]
> y
did you create link 'new_class' of object type 'default::Character'?
[y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00002.edgeql, id:
m1uttd6f7fpiwiwikhdh6qyijb6pcji747ccg2cyt5357i3wsj313q
$ edgedb migrate
Applied m1uttd6f7fpiwiwikhdh6qyijb6pcji747ccg2cyt5357i3wsj313q
(00002.edgeql)
```

It makes sense to add a data migration as a way of consistently creating `NewClass` objects as well as populating `new_class` links based on the existing `class` property. So we first create an empty migration:

```
$ edgedb migration create --allow-empty
Created ./dbschema/migrations/00003.edgeql, id:
m1iztxroh3ifoeqmvxncy77whnaei6tp5j3sewyxtrfyrsronjkxgga
```

And then edit the `00003.edgeql` file to create and update objects:

```
CREATE MIGRATION m1iztxroh3ifoeqmvxncy77whnaei6tp5j3sewyxtrfyrsronjkxgga
    ONTO m1uttd6f7fpiwiwikhdh6qyijb6pcji747ccg2cyt5357i3wsj313q
{
    insert default::NewClass {
        name := 'Warrior',
        skills := {'punch', 'kick', 'run', 'jump'},
    };
    insert default::NewClass {
        name := 'Scholar',
        skills := {'read', 'write', 'analyze', 'refine'},
    };
    insert default::NewClass {
        name := 'Rogue',
        skills := {'impress', 'sing', 'steal', 'run', 'jump'},
    };

    update default::Character
    set {
        new_class := assert_single(
            select default::NewClass
            filter .name ilike <str>default::Character.class
        ),
    };
};
```

Trying to apply the data migration will produce the following reminder:

```
$ edgedb migrate
edgedb error: could not read migrations in ./dbschema/migrations:
could not read migration file ./dbschema/migrations/00003.edgeql:
migration name should be
`m1e3d3eg3j2pr7acie4n5rrhaddyhkiy5kgckd517h5ysrpgwx15a` but
`m1iztxroh3ifoeqmvxncy77whnaei6tp5j3sewyxtrfysonjkxgga` is used
instead.
Migration names are computed from the hash of the migration
contents. To proceed you must fix the statement to read as:
  CREATE MIGRATION m1e3d3eg3j2pr7acie4n5rrhaddyhkiy5kgckd517h5ysrpgwx15a
    ONTO ...
if this migration is not applied to any database. Alternatively,
revert the changes to the file.
```

The migration tool detected that we've altered the file and asks us to update the migration name (acting as a checksum) if this was deliberate. This is done as a precaution against accidental changes. Since we've done this on purpose, we can update the file and run `edgedb migrate` again.

We can see the changes after the data migration is complete:

```
db> select Character {
...     name,
...     class,
...     new_class: {
...         name,
...     }
... };
{
    default::Character {
        name: 'Alice',
        class: warrior,
        new_class: default::NewClass {name: 'Warrior'},
    },
    default::Character {
        name: 'Billie',
        class: scholar,
        new_class: default::NewClass {name: 'Scholar'},
    },
    default::Character {
        name: 'Cameron',
        class: rogue,
        new_class: default::NewClass {name: 'Rogue'},
    },
}
```

Everything seems to be in order. It is time to clean up the old property and `CharacterClass` `enum`:

```
type NewClass {
    required property name -> str;
    multi property skills -> str;
}
```

(continues on next page)

(continued from previous page)

```
type Character {
    required property name -> str;
    link new_class -> NewClass;
}
```

```
type NewClass {
    required name: str;
    multi skills: str;
}

type Character {
    required name: str;
    new_class: NewClass;
}
```

The migration tools should have no trouble detecting the things we just removed:

```
$ edgedb migration create
did you drop property 'class' of object type 'default::Character'?
[y,n,l,c,b,s,q,?]
> y
did you drop scalar type 'default::CharacterClass'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00004.edgeql, id:
m1jdnz5bxjj6kjz2pylvudli5rvw4jyr2ilpb4hit3yutwi3bq34ha
$ edgedb migrate
Applied m1jdnz5bxjj6kjz2pylvudli5rvw4jyr2ilpb4hit3yutwi3bq34ha
(00004.edgeql)
```

Now that the original property and scalar type are gone, we can rename the “new” components, so that they become class link and CharacterClass type, respectively:

```
type CharacterClass {
    required property name -> str;
    multi property skills -> str;
}
```

```
type Character {
    required property name -> str;
    link class -> CharacterClass;
}
```

```
type CharacterClass {
    required name: str;
    multi skills: str;
}
```

```
type Character {
    required name: str;
    class: CharacterClass;
}
```

The migration tools pick up the changes without any issues again. It may seem tempting to combine the last two steps,

but deleting and renaming in a single step would cause the migration tools to report a name clash. As a general rule, it is a good idea to never mix renaming and deleting of closely interacting entities in the same migration.

```
$ edgedb migration create
did you rename object type 'default::NewClass' to
'default::CharacterClass'? [y,n,l,c,b,s,q,?]
> y
did you rename link 'new_class' of object type 'default::Character' to
'class'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00005.edgeql, id:
m1ra4fhx2erkygbhi7qjxt27yup5aw5hkr5bekn5y5jeam5yn57vs
$ edgedb migrate
Applied m1ra4fhx2erkygbhi7qjxt27yup5aw5hkr5bekn5y5jeam5yn57vs
(00005.edgeql)
```

Finally, we have replaced the original `class` property with a link:

```
db> select Character {
...     name,
...     class: {
...         name,
...         skills,
...     }
... };
{
    default::Character {
        name: 'Alice',
        class: default::CharacterClass {
            name: 'Warrior',
            skills: {'punch', 'kick', 'run', 'jump'},
        },
    },
    default::Character {
        name: 'Billie',
        class: default::CharacterClass {
            name: 'Scholar',
            skills: {'read', 'write', 'analyze', 'refine'},
        },
    },
    default::Character {
        name: 'Cameron',
        class: default::CharacterClass {
            name: 'Rogue',
            skills: {'impress', 'sing', 'steal', 'run', 'jump'},
        },
    },
}
```

### 4.3.5 Adding a required link

This example shows how to setup a required link. We'll use a character in an adventure game as the type of data we will evolve.

Let's start with this schema:

```
type Character {
    required property name -> str;
}
```

```
type Character {
    required name: str;
}
```

We edit the schema file and perform our first migration:

```
$ edgedb migration create
did you create object type 'default::Character'? [y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00001.edgeql, id:
m1xvu7o4z5f5xfwuun2vee2cryvvzh5lfilwgkulmqpifo5m3dnd6a
$ edgedb migrate
Applied m1xvu7o4z5f5xfwuun2vee2cryvvzh5lfilwgkulmqpifo5m3dnd6a
(00001.edgeql)
```

This time around let's practice performing a data migration and set up our character data. For this purpose we can create an empty migration and fill it out as we like:

```
$ edgedb migration create --allow-empty
Created ./dbschema/migrations/00002.edgeql, id:
m1lclvwdpwitjj4xqm45wp74y4wfyadljct5o6bsctlh5xbto74iq
```

We edit the `00002.edgeql` file by simply adding the query to add characters to it. We can use [for](#) to add multiple characters like this:

```
CREATE MIGRATION m1lclvwdpwitjj4xqm45wp74y4wfyadljct5o6bsctlh5xbto74iq
    ONTO m1xvu7o4z5f5xfwuun2vee2cryvvzh5lfilwgkulmqpifo5m3dnd6a
{
+    for name in {'Alice', 'Billie', 'Cameron', 'Dana'}
+    union (
+        insert default::Character {
+            name := name
+        }
+    );
};
```

Trying to apply the data migration will produce the following reminder:

```
$ edgedb migrate
edgedb error: could not read migrations in ./dbschema/migrations:
could not read migration file ./dbschema/migrations/00002.edgeql:
migration name should be
`m1juin65wriqmb4vvg23fiyajjxlzj2jyjv5qp36uxenit5y63g2iq` but
```

(continues on next page)

(continued from previous page)

```
`m1lclvwdpwitjj4xqm45wp74y4wjiyadljct5o6bsctlnh5xbto74iq` is used instead.
```

Migration names are computed from the hash of the migration contents. To proceed you must fix the statement to read as:

```
CREATE MIGRATION m1juin65wriqmb4vwg23fiyajjxlzj2jyjv5qp36uxenit5y63g2iq
ONTO ...
```

if this migration is not applied to any database. Alternatively, revert the changes to the file.

The migration tool detected that we've altered the file and asks us to update the migration name (acting as a checksum) if this was deliberate. This is done as a precaution against accidental changes. Since we've done this on purpose, we can update the file and run `edgedb migrate` again.

```
- CREATE MIGRATION m1lclvwdpwitjj4xqm45wp74y4wjiyadljct5o6bsctlnh5xbto74iq
+ CREATE MIGRATION m1juin65wriqmb4vwg23fiyajjxlzj2jyjv5qp36uxenit5y63g2iq
    ONTO m1xvu7o4z5f5xfwuun2vee2cryvvzh5lfilwgkulmqpifo5m3dnd6a
{
    # ...
};
```

After we apply the data migration we should be able to see the added characters:

```
db> select Character {name};
{
    default::Character {name: 'Alice'},
    default::Character {name: 'Billie'},
    default::Character {name: 'Cameron'},
    default::Character {name: 'Dana'},
}
```

Let's add a character class represented by a new type to our schema and data. Unlike in [this scenario](#), we will add the required link class right away, without any intermediate properties. So we end up with a schema like this:

```
type CharacterClass {
    required property name -> str;
    multi property skills -> str;
}

type Character {
    required property name -> str;
    required link class -> CharacterClass;
}
```

```
type CharacterClass {
    required name: str;
    multi skills: str;
}

type Character {
    required name: str;
    required class: CharacterClass;
}
```

We go ahead and try to apply this new schema:

```
$ edgedb migration create
did you create object type 'default::CharacterClass'? [y,n,l,c,b,s,q,?]
> y
did you create link 'class' of object type 'default::Character'?
[y,n,l,c,b,s,q,?]
> y
Please specify an expression to populate existing objects in order to make
link 'class' of object type 'default::Character' required:
fill_expr>
```

Uh-oh! Unlike in a situation with a *required property*, it's not a good idea to just *insert* a new `CharacterClass` object for every character. So we should abort this migration attempt and rethink our strategy. We need a separate step where the `class` link is not *required* so that we can write some custom queries to handle the character classes:

```
type CharacterClass {
    required property name -> str;
    multi property skills -> str;
}

type Character {
    required property name -> str;
    link class -> CharacterClass;
}
```

```
type CharacterClass {
    required name: str;
    multi skills: str;
}

type Character {
    required name: str;
    class: CharacterClass;
}
```

We can now create a migration for our new schema, but we won't apply it right away:

```
$ edgedb migration create
did you create object type 'default::CharacterClass'? [y,n,l,c,b,s,q,?]
> y
did you create link 'class' of object type 'default::Character'?
[y,n,l,c,b,s,q,?]
> y
Created ./dbschema/migrations/00003.edgeql, id:
m1jie3xamsm2b7ygqccwfh2degdi45oc7mwuyzjkanh2qwgiqvi2ya
```

We don't need to create a blank migration to add data, we can add our modifications into the migration that adds the `class` link directly. Doing this makes sense when the schema changes seem to require the data migration and the two types of changes logically go together. We will need to create some `CharacterClass` objects as well as *update* the `class` link on existing `Character` objects:

```
CREATE MIGRATION m1jie3xamsm2b7ygqccwfh2degdi45oc7mwuyzjkanh2qwgiqvi2ya
    ONTO m1juin65wriqmb4vvg23fiyajjxlzj2jyjv5qp36uxenit5y63g2iq
{
```

(continues on next page)

(continued from previous page)

```

CREATE TYPE default::CharacterClass {
    CREATE REQUIRED PROPERTY name -> std::str;
    CREATE MULTI PROPERTY skills -> std::str;
};

ALTER TYPE default::Character {
    CREATE LINK class -> default::CharacterClass;
};

+ insert default::CharacterClass {
+     name := 'Warrior',
+     skills := {'punch', 'kick', 'run', 'jump'},
+ };
+ insert default::CharacterClass {
+     name := 'Scholar',
+     skills := {'read', 'write', 'analyze', 'refine'},
+ };
+ insert default::CharacterClass {
+     name := 'Rogue',
+     skills := {'impress', 'sing', 'steal', 'run', 'jump'},
+ };
+ # All warriors
+ update default::Character
+ filter .name in {'Alice'}
+ set {
+     class := assert_single(
+         select default::CharacterClass
+         filter .name = 'Warrior'
+     ),
+ };
+ # All scholars
+ update default::Character
+ filter .name in {'Billie'}
+ set {
+     class := assert_single(
+         select default::CharacterClass
+         filter .name = 'Scholar'
+     ),
+ };
+ # All rogues
+ update default::Character
+ filter .name in {'Cameron', 'Dana'}
+ set {
+     class := assert_single(
+         select default::CharacterClass
+         filter .name = 'Rogue'
+     ),
+ };
+ };
};

```

In a real game we might have a lot more characters and so a good way to update them all is to update characters of the same class in bulk.

Just like before we'll be reminded to fix the migration name since we've altered the migration file. After fixing the

migration hash we can apply it. Now all our characters should have been assigned their classes:

```
db> select Character {
...     name,
...     class: {
...         name
...     }
... };
{
    default::Character {
        name: 'Alice',
        class: default::CharacterClass {name: 'Warrior'},
    },
    default::Character {
        name: 'Billie',
        class: default::CharacterClass {name: 'Scholar'},
    },
    default::Character {
        name: 'Cameron',
        class: default::CharacterClass {name: 'Rogue'},
    },
    default::Character {
        name: 'Dana',
        class: default::CharacterClass {name: 'Rogue'},
    },
}
```

We're finally ready to make the `class` link *required*. We update the schema:

```
type CharacterClass {
    required property name -> str;
    multi property skills -> str;
}

type Character {
    required property name -> str;
    required link class -> CharacterClass;
}
```

```
type CharacterClass {
    required name: str;
    multi skills: str;
}

type Character {
    required name: str;
    required class: CharacterClass;
}
```

And we perform our final migration:

```
$ edgedb migration create
did you make link 'class' of object type 'default::Character' required?
[y,n,l,c,b,s,q,?]
```

(continues on next page)

(continued from previous page)

```
> y
Please specify an expression to populate existing objects in order to
make link 'class' of object type 'default::Character' required:
fill_expr> assert_exists(.class)
Created ./dbschema/migrations/00004.edgeql, id:
m14yblybdo77c7bjtm6nugiy5cs6pl6rnuzo5b27gamy4zhuwjifia
```

The migration system doesn't know that we've already assigned `class` values to all the `Character` objects, so it still asks us for an expression to be used in case any of the objects need it. We can use `assert_exists(.class)` here as a way of being explicit about the fact that we expect the values to already be present. Missing values would have caused an error even without the `assert_exists` wrapper, but being explicit may help us capture the intent and make debugging a little easier if anyone runs into a problem at this step.

In fact, before applying this migration, let's actually add a new `Character` to see what happens:

```
db> insert Character {name := 'Eric'};
{
    default::Character {
        id: 9f4ac7a8-ac38-11ec-b076-afefd12d7e66,
    },
}
```

Our attempt at migrating fails as we expected:

```
$ edgedb migrate
edgedb error: MissingRequiredError: missing value for required link
'class' of object type 'default::Character'
  Detail: Failing object id is 'ee604992-c1b1-11ec-ad59-4f878963769f'.
```

After removing the bugged `Character`, we can migrate without any problems:

```
$ edgedb migrate
Applied m14yblybdo77c7bjtm6nugiy5cs6pl6rnuzo5b27gamy4zhuwjifia
(00004.edgeql)
```

## 4.4 Cheatsheets

### `edb-alt-title` Cheatsheets: EdgeDB by example

Just getting started? Keep an eye on this collection of cheatsheets with handy examples for what you'll need to get started with EdgeDB. After familiarizing yourself with them, feel free to dive into more EdgeDB via our longer [interactive tutorial](#) and **much** longer [Easy EdgeDB textbook](#).

EdgeQL:

- `select` – Retrieve or compute a set of values.
- `insert` – Create new database objects.
- `update` – Update database objects.
- `delete` – Remove objects from the database.
- `GraphQL` – GraphQL queries supported natively out of the box.

Schema:

- *Object Types* – Make your own object and abstract types on top of existing system types.
- *User Defined Functions* – Write and overload your own strongly typed functions.
- *Expression Aliases* – Use aliases to create new types and modify existing ones on the fly.
- *Schema Annotations* – Add human readable descriptions to items in your schema.

CLI/Admin:

- *CLI Usage* – Getting your database started.
- *Interactive Shell* – Shortcuts for frequently used commands in the EdgeDB Interactive Shell.
- *Administration* – Database and role creation, passwords, port configuration, etc.

#### 4.4.1 Selecting data

---

**Note:** The types used in these queries are defined [here](#).

---

Select a Movie with associated actors and reviews with their authors:

```
select Movie {
    id,
    title,
    year,
    description,

    actors: {
        id,
        full_name,
    },

    reviews := .<movie[is Review] {
        id,
        body,
        rating,
        author: {
            id,
            name,
        }
    },
}
filter .id = <uuid>'09c34154-4148-11ea-9c68-5375ca908326'
```

---

Select movies with Keanu Reeves:

```
select Movie {
    id,
    title,
    year,
    description,
```

(continues on next page)

(continued from previous page)

```
}
filter .actors.full_name = 'Keanu Reeves'
```

Select all actors that share the last name with other actors and include the same-last-name actor list as well:

```
select Person {
    id,
    full_name,
    same_last_name := (
        with
            P := detached Person
        select P {
            id,
            full_name,
        }
        filter
            # same last name
            P.last_name = Person.last_name
            and
            # not the same person
            P != Person
        ),
    }
filter exists .same_last_name
```

The same query can be refactored moving the `with` block to the top-level:

```
with
    # don't need detached at top-level
    P := Person
select Person {
    id,
    full_name,
    same_last_name := (
        select P {
            id,
            full_name,
        }
        filter
            # same last name
            P.last_name = Person.last_name
            and
            # not the same person
            P != Person
        ),
    }
filter exists .same_last_name
```

Select user names and the number of reviews they have:

```
select (
    User.name,
    count(User.<author[is Review])
)
```

For every user and movie combination, select whether the user has reviewed the movie (beware, in practice this maybe a very large result):

```
select (
    User.name,
    Movie.title,
    Movie in User.<author[is Review].movie
)
```

Perform a set intersection of all actors with all directors:

```
with
    # get the set of actors and set of directors
    Actor := Movie.actors,
    Director := Movie.director,
    # set intersection is done via the filter clause
    select Actor filter Actor in Director;
```

To order a set of scalars first assign the set to a variable and use the variable in the order by clause.

```
select numbers := {3, 1, 2} order by numbers;

# alternatively
with numbers := {3, 1, 2}
select numbers order by numbers;
```

Selecting free objects.

It is also possible to package data into a *free object*. *Free objects* are meant to be transient and used either to more efficiently store some intermediate results in a query or for re-shaping the output. The advantage of using *free objects* over *tuples* is that it is easier to package data that potentially contains empty sets as links or properties of the *free object*. The underlying type of a *free object* is `std::FreeObject`.

Consider the following query:

```
with U := (select User filter .name like '%user%')
select {
    matches := U {name},
    total := count(U),
    total_users := count(User),
};
```

The `matches` are potentially `{}`, yet the query will always return a single *free object* with `results`, `total`, and `total_users`. To achieve the same using a *named tuple*, the query would have to be modified like this:

```
with U := (select User filter .name like '%user%')
select (
    matches := array_agg(U {name}),
    total := count(U),
    total_users := count(User),
);
```

Without the `array_agg()` the above query would return `{}` instead of the named tuple if no `matches` are found.

See also
<a href="#">EdgeQL &gt; Select</a>
<a href="#">Reference &gt; Commands &gt; Select</a>
<a href="#">Tutorial &gt; Basic Queries &gt; Objects</a>
<a href="#">Tutorial &gt; Basic Queries &gt; Filters</a>
<a href="#">Tutorial &gt; Basic Queries &gt; Aggregates</a>
<a href="#">Tutorial &gt; Nested Structures &gt; Shapes</a>
<a href="#">Tutorial &gt; Nested Structures &gt; Polymorphism</a>

#### 4.4.2 Inserting data

---

**Note:** The types used in these queries are defined [here](#).

---

Insert basic movie stub:

```
insert Movie {
    title := 'Dune',
    year := 2020,
    image := 'dune2020.jpg',
    directors := (
        select Person
        filter
            .full_name = 'Denis Villeneuve'
    )
}
```

---

Alternatively, insert a movie using JSON input value:

```
with
    # Cast the JSON $input into a tuple, which we will
    # use to populate the Person record.
    data := <tuple<
        title: str,
        year: int64,
        image: str,
        directors: array<str>,
        actors: array<str>
    >> <json>$input
```

(continues on next page)

(continued from previous page)

```
insert Movie {
    title := data.title,
    year := data.year,
    image := data.image,
    directors := (
        select Person
        filter
            .full_name in array_unpack(data.directors)
    ),
    actors := (
        select Person
        filter
            .full_name in array_unpack(data.actors)
    )
}
```

Insert several nested objects at once:

```
# Create a new review and a new user in one step.
insert Review {
    body := 'Dune is cool',
    rating := 5,
    # The movie record already exists, so select it.
    movie := (
        select Movie
        filter
            .title = 'Dune'
            and
            .year = 2020
        # the limit is needed to satisfy the single
        # link requirement validation
        limit 1
    ),
    # This is a new user, so insert one.
    author := (
        insert User {
            name := 'dune_fan_2020',
            image := 'default_avatar.jpg',
        }
    )
}
```

Sometimes it's necessary to check whether some object exists and create it if it doesn't. If this type of object has an exclusive property, the `unless conflict` clause can make the `insert` command idempotent. So running such a command would guarantee that a copy of the object exists without the need for more complex logic:

```
# Try to create a new User
insert User {
    name := "Alice",
```

(continues on next page)

(continued from previous page)

```

    image := "default_avatar.jpg",
}
# and do nothing if a User with this name already exists
unless conflict

```

If more than one property is exclusive, it is possible to specify which one of them is considered when a conflict is detected:

```

# Try to create a new User
insert User {
    name := "Alice",
    image := "default_avatar.jpg",
}
# and do nothing if a User with this name already exists
unless conflict on .name

```

“Upserts” can be performed by using the `unless conflict` clause and specifying what needs to be updated:

```

select (
    # Try to create a new User,
    insert User {
        name := "Alice",
        image := "my_face.jpg",
    }

    # but if a User with this name already exists,
    unless conflict on .name
    else (
        # update that User's record instead.
        update User
        set {
            image := "my_face.jpg"
        }
    )
) {
    name,
    image
}

```

Rather than acting as an “upsert”, the `unless conflict` clause can be used to insert or select an existing record, which is handy for inserting nested structures:

```

# Create a new review and a new user in one step.
insert Review {
    body := 'Loved it!!!',
    rating := 5,
    # The movie record already exists, so select it.
    movie := (
        select Movie
        filter

```

(continues on next page)

(continued from previous page)

```

.title = 'Dune'
and
.year = 2020
# the limit is needed to satisfy the single
# link requirement validation
limit 1
),

# This might be a new user or an existing user. Some
# other part of the app handles authentication, this
# endpoint is used as a generic way to post a review.
author := (
    # Try to create a new User,
    insert User {
        name := "dune_fan_2020",
        image := "default_avatar.jpg",
    }

    # but if a User with this name already exists,
    unless conflict on .name
    # just pick that existing User as the author.
    else User
)
}

```

See also
<a href="#">EdgeQL &gt; Insert</a>
<a href="#">Reference &gt; Commands &gt; Insert</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Insert</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Upsert</a>

#### 4.4.3 Updating data

---

**Note:** The types used in these queries are defined [here](#).

---

Flag all reviews to a specific movie:

```

update Review
filter
    Review.movie.title = 'Dune'
    and
    Review.movie.director.last_name = 'Villeneuve'
set {
    flag := True
}

```

---

Add an actor with a specific `list_order` link property to a movie:

```
update Movie
filter
    .title = 'Dune'
    and
    .directors.last_name = 'Villeneuve'
set {
    actors := (
        insert Person {
            first_name := 'Timothee',
            last_name := 'Chalamet',
            image := 'tchalamet.jpg',
            @list_order := 1,
        }
    )
}
```

---

Using a `for` query to set a specific `list_order` link property for the actors list:

```
update Movie
filter
    .title = 'Dune'
    and
    .directors.last_name = 'Villeneuve'
set {
    actors := (
        for x in {
            ('Timothee Chalamet', 1),
            ('Zendaya', 2),
            ('Rebecca Ferguson', 3),
            ('Jason Momoa', 4),
        }
        union (
            select Person {@list_order := x.1}
            filter .full_name = x.0
        )
    )
}
```

---

Updating a multi link by adding one more item:

```
update Movie
filter
    .title = 'Dune'
    and
    .directors.last_name = 'Villeneuve'
set {
    actors += (
        insert Person {
            first_name := 'Dave',
            last_name := 'Bautista',
        }
    )
}
```

(continues on next page)

(continued from previous page)

```

        image := 'dbautista.jpg',
    }
}
}
```

Updating a multi link by removing an item:

```

update Movie
filter
    .title = 'Dune'
    and
    .directors.last_name = 'Villeneuve'
set {
    actors -= (
        select Person
        filter
            .full_name = 'Jason Momoa'
    )
}
```

Update the `list_order` link property for a specific link:

```

update Movie
filter
    .title = 'Dune'
    and
    .directors.last_name = 'Villeneuve'
set {
    # The += operator will allow updating only the
    # specified actor link.
    actors += (
        select Person {
            @list_order := 5,
        }
        filter .full_name = 'Jason Momoa'
    )
}
```

#### See also

[EdgeQL > Update](#)

[Reference > Commands > Update](#)

[Tutorial > Data Mutations > Update](#)

#### 4.4.4 Deleting data

---

**Note:** The types used in these queries are defined [here](#).

---

Delete all reviews from a specific user:

```
delete Review
filter .author.name = 'trouble2020'
```

---

Alternative way to delete all reviews from a specific user:

```
delete (
    select User
    filter .name = 'troll2020'
).<author[is Review]
```

See also
<a href="#">EdgeQL &gt; Delete</a>
<a href="#">Reference &gt; Commands &gt; Delete</a>
<a href="#">Tutorial &gt; Data Mutations &gt; Delete</a>

#### 4.4.5 Using link properties

##### index property

Links can contain **properties**. These are distinct from links themselves (which we refer to as simply “links”) and are used to store metadata about a link. Due to how they’re persisted under the hood, link properties have a few additional constraints: they’re always *single* and *optional*.

---

**Note:** In practice, link properties are best used with many-to-many relationships (`multi` links without any exclusive constraints). For one-to-one, one-to-many, and many-to-one relationships the same data should be stored in object properties instead.

---

##### 4.4.5.1 Declaration

Let’s create a `Person.friends` link with a `strength` property corresponding to the strength of the friendship.

```
type Person {
    required property name -> str { constraint exclusive };

    multi link friends -> Person {
        property strength -> float64;
    }
}
```

```
type Person {
    required name: str { constraint exclusive };

    multi friends: Person {
        strength: float64;
    }
}
```

#### 4.4.5.2 Constraints

```
type Person {
    required property name -> str { constraint exclusive };

    multi link friends -> Person {
        property strength -> float64;
        constraint expression on (
            __subject__@strength >= 0
        );
    }
}
```

```
type Person {
    required name: str { constraint exclusive };

    multi friends: Person {
        strength: float64;
        constraint expression on (
            __subject__@strength >= 0
        );
    }
}
```

#### 4.4.5.3 Indexes

To index on a link property, you must declare an abstract link and extend it.

```
abstract link friendship {
    property strength -> float64;
    index on (__subject__@strength);
}

type Person {
    required property name -> str { constraint exclusive };
    multi link friends extending friendship -> Person;
}
```

```
abstract link friendship {
    strength: float64;
    index on (__subject__@strength);
}
```

(continues on next page)

(continued from previous page)

```
type Person {
    required name: str { constraint exclusive };
    multi friends: Person {
        extending friendship;
    };
}
```

#### 4.4.5.4 Inserting

The `@strength` property is specified in the `shape` of the `select` subquery. This is only valid in a subquery *inside* an `insert` statement.

```
insert Person {
    name := "Bob",
    friends := (
        select detached Person {
            @strength := 3.14
        }
        filter .name = "Alice"
    )
}
```

**Note:** We are using the `detached` operator to unbind the `Person` reference from the scope of the `insert` query.

When doing a nested insert, link properties can be directly included in the inner `insert` subquery.

```
insert Person {
    name := "Bob",
    friends := (
        insert Person {
            name := "Jane",
            @strength := 3.14
        }
    )
}
```

#### 4.4.5.5 Updating

```
update Person
filter .name = "Bob"
set {
    friends += (
        select .friends {
            @strength := 3.7
        }
        filter .name = "Alice"
    )
};
```

The example updates the `@strength` property of Bob's friends link to 3.7.

In the context of multi links the `+=` operator works like an an insert/update operator.

To update one or more links in a multi link, you can select from the current linked objects, as the example does. Use a detached selection if you want to insert/update a wider selection of linked objects instead.

#### 4.4.5.6 Querying

```
edgedb> select Person {
.....   friends: {
.....     name,
.....     @strength
.....   }
..... };
{
  default::Person {name: 'Alice', friends: {}},
  default::Person {
    name: 'Bob',
    friends: {
      default::Person {name: 'Alice', @strength: 3.7}
    }
  },
}
```

---

**Note:** Specifying link properties of a computed backlink in your shape is supported as of EdgeDB 3.0.

If you have this schema:

```
type Person {
  required name: str;
  multi follows: Person {
    followed: datetime {
      default := datetime_of_statement();
    };
  };
  multi link followers := .<follows[is Person];
}
```

this query will work as of EdgeDB 3.0:

```
select Person {
  name,
  followers: {
    name,
    @followed
  }
};
```

even though `@followed` is a link property of `follows` and we are accessing is through the computed backlink `followers` instead.

If you need link properties on backlinks in earlier versions of EdgeDB, you can use this workaround:

```
select Person {
    name,
    followers := .<follows[is Person] {
        name,
        followed := @followed
    }
};
```

See also
<a href="#">Data Model &gt; Links &gt; Link properties</a>
<a href="#">SDL &gt; Properties</a>
<a href="#">DDL &gt; Properties</a>
<a href="#">Introspection &gt; Object Types</a>

#### 4.4.6 Working with booleans

Boolean expressions can be tricky sometimes, so here are a handful of tips and gotchas.

---

There's a fundamental difference in how `{}` is treated by `and` and `or` vs `all()` and `any()`. The operators `and` and `or` require both operands to produce a result, which means that an `{}` as one of the inputs necessarily produces an `{}` as the output:

```
db> select false and <bool>{};
{}
db> select true or <bool>{};
{}
```

The functions `all()` and `any()`, however, produce a result for all possible input sets, regardless of the number of elements:

```
db> select all({false, {}});
{false}
db> select any({true, {}});
{true}
```

Note that expressions like `{false, {}}` are equivalent to `{false}` and so the above are just generalizations of boolean operators `and` and `or` to a set of 1 element. So the result for 1 element is fairly intuitive. However, the results produced by these functions for `{}` may be surprising (even though they are mathematically consistent):

```
db> select all(<bool>{});
{true}
db> select any(<bool>{});
{false}
```

---

There's no direct analogue to the boolean operator "short-circuiting" that's implemented in many other languages because in EdgeQL the order of evaluation of subexpressions is generally not defined. However, there are expressions that achieve the same end goal for which "short-circuiting" is used.

---

The most basic filtering doesn't even require any "short-circuiting" guards because these are already implied by EdgeQL. For example, "*get all accounts that completed 5 steps of the process*":

```
select Account filter .steps = 5;
```

When there's a need to express that a field is initialized, but not equal to some particular value "short-circuiting" is often used to discard non-initialized values (e.g. `acc.steps` is not `None` and `acc.steps != 5`). This is another case where EdgeQL doesn't require any additional guards. For example "*get all initialized accounts that have not completed 5 steps of the process*":

```
select Account filter .steps != 5;
```

If the task boils down to annotating every element as opposed to selecting specific ones, the use of `?=` instead of the plain `=` helps to deal with optional properties. For example, "*get all accounts and annotate them with their completeness status*":

```
select Account {
    completed := .steps ?= 5
};
```

Sometimes the condition that needs to be evaluated is not a simple equality comparison. The `??` can help out in these cases. For example, "*get all accounts and annotate them on whether or not they are half-way completed*":

```
select Account {
    completed := (.steps > 2) ?? false
};
```

The above trick can also be useful for filtering based on some boolean condition that's not just a plain equality. For example, "*get only the accounts that are less than half-way completed*":

```
select Account {
    too_few_steps := (.steps <= 2) ?? true
} filter .too_few_steps;
```

The above will end up including the computed flag `too_few_steps` in the output, but this is sometimes undesirable. In order to avoid including it, the query can be refactored like this:

```
select Account {
    name,
    email,
    # whatever other relevant data is needed
} filter (.steps <= 2) ?? true;
```

When using `?=`, `?=`, or `??` it is important to keep in mind how they interact with *path expressions* that can sometimes be `{}`. Basically, these operators don't actually affect the path expression, they only act on the *results* of the path expression. Consider the following two queries:

```
select Account {
    too_few_steps := (.steps <= 2) ?? true
}.too_few_steps;

select (Account.steps <= 2) ?? true;
```

The first query is going to output `true` or `false` for every account, based on the specified criteria. It's important to note that the number of the results is going to be exactly the same as the number of the accounts in the system. The second query may look like a more compact version of the first query, but it behaves completely differently. If all of the account are “uninitialized” (`steps := {}`) or there are no accounts at all, it will produce a single result `true`. That's because the expression `Account.steps <= 2` produces an empty set in this case and so the `??` returns the second operand. On the other hand, if there are any accounts with some concrete number of `steps`, then the expression `Account.steps <= 2` will produce a result for *those accounts only*. The `??` won't change that result because the result is already non-empty and so no coalescing will take place.

Computed in shapes get evaluated *for each object*, whereas path expressions only produce as many values as are *reachable* by the path. So when all objects must be considered, computed links and properties in shapes are a good way to handle complex expressions or filters. When only objects with specific properties are relevant, path expressions are a good compact way of handling this.

---

There's also another way to evaluate something on a per-object basis and that's by using a `for` query. For example, let's rewrite the query that outputs `true` or `false` for every account, based on the number of completed steps:

```
for A in Account
union (A.steps <= 2) ?? true;
```

---

Expressions specified in shapes, `for`, or `filter` clauses are all evaluated on a per-item basis. The gotchas in these cases can arise from using longer path expressions combined with `??`, `?=`, or `?!=`. For example, let's say that in addition to accounts and steps we also have different “projects” with a multi-link of accounts marking progress in them. So keeping that in mind, let's try writing a query to “*get all projects that have linked accounts which made little progress (fewer than 3 ‘steps’)*”:

```
select Project
filter .accounts.steps < 3;
```

Well, that's not right. Projects that have accounts without any `steps` of progress are not reported by the above query. So maybe adding a `??` will help?

```
select Project
filter (.accounts.steps < 3) ?? true;
```

This is better as the results now include projects where none of the accounts made any progress. However, any project that has a mix of accounts that made more than 2 steps of progress and accounts that haven't even started is still missing from the results. So we can either use the trick we used before with shapes or we can add another `for` subquery:

```
select Project
filter (
    for A in .accounts
    union (A.steps < 3) ?? true
);
```

---

Note that the `filter` clause behaves as an implicit `any()`. This means that the following are semantically equivalent:

```
select User
filter .friends.name = 'Alice';

select User
filter any(.friends.name = 'Alice');
```

#### 4.4.7 Object types

Define an abstract type:

```
abstract type HasImage {
    # just a URL to the image
    required property image -> str;
    index on (.image);
}
```

```
abstract type HasImage {
    # just a URL to the image
    required image: str;
    index on (.image);
}
```

Define a type extending from the abstract:

```
type User extending HasImage {
    required property name -> str {
        # Ensure unique name for each User.
        constraint exclusive;
    }
}
```

```
type User extending HasImage {
    required name: str {
        # Ensure unique name for each User.
        constraint exclusive;
    }
}
```

Define a type with constraints and defaults for properties:

```
type Review {
    required property body -> str;
    required property rating -> int64 {
        constraint min_value(0);
        constraint max_value(5);
    }
    required property flag -> bool {
        default := False;
    }
}
```

(continues on next page)

(continued from previous page)

```

}

required link author -> User;
required link movie -> Movie;

required property creation_time -> datetime {
    default := datetime_current();
}
}
```

```

type Review {
    required body: str;
    required rating: int64 {
        constraint min_value(0);
        constraint max_value(5);
    }
    required flag: bool {
        default := False;
    }

    required author: User;
    required movie: Movie;

    required creation_time: datetime {
        default := datetime_current();
    }
}
```

Define a type with a property that is computed from the combination of the other properties:

```

type Person extending HasImage {
    required property first_name -> str {
        default := '';
    }
    required property middle_name -> str {
        default := '';
    }
    required property last_name -> str;
    property full_name :=
        (
            (
                (.first_name ++ ' ')
                if .first_name != '' else
                ''
            ) ++
            (
                (.middle_name ++ ' ')
                if .middle_name != '' else
                ''
            ) ++
        )
}
```

(continues on next page)

(continued from previous page)

```

        .last_name
    );
property bio -> str;
}

```

```

type Person extending HasImage {
    required first_name: str {
        default := '';
    }
    required middle_name: str {
        default := '';
    }
    required last_name: str;
    property full_name := (
        (
            (.first_name ++ ' ')
            if .first_name != '' else
            ''
        ) ++
        (
            (.middle_name ++ ' ')
            if .middle_name != '' else
            ''
        ) ++
        .last_name
    );
    bio: str;
}

```

Define an abstract links:

```

abstract link crew {
    # Provide a way to specify some "natural"
    # ordering, as relevant to the movie. This
    # may be order of importance, appearance, etc.
property list_order -> int64;
}

abstract link directors extending crew;

abstract link actors extending crew;

```

```

abstract link crew {
    # Provide a way to specify some "natural"
    # ordering, as relevant to the movie. This
    # may be order of importance, appearance, etc.
    list_order: int64;
}

```

(continues on next page)

(continued from previous page)

```
abstract link directors {
    extending crew;
};

abstract link actors {
    extending crew;
};
```

Define a type using abstract links and a computed property that aggregates values from another linked type:

```
type Movie extending HasImage {
    required property title -> str;
    required property year -> int64;

    # Add an index for accessing movies by title and year,
    # separately and in combination.
    index on (.title);
    index on (.year);
    index on ((.title, .year));

    property description -> str;

    multi link directors extending crew -> Person;
    multi link actors extending crew -> Person;

    property avg_rating := math::mean(.<movie[is Review].rating);
}
```

```
type Movie extending HasImage {
    required title: str;
    required year: int64;

    # Add an index for accessing movies by title and year,
    # separately and in combination.
    index on (.title);
    index on (.year);
    index on ((.title, .year));

    description: str;

    multi directors: Person {
        extending crew;
    };
    multi actors: Person {
        extending crew
    };

    property avg_rating := math::mean(.<movie[is Review].rating);
}
```

Define an *auto-incrementing* scalar type and an object type using it as a property:

```
scalar type TicketNo extending sequence;

type Ticket {
    property number -> TicketNo {
        constraint exclusive;
    }
}
```

```
scalar type TicketNo extending sequence;

type Ticket {
    number: TicketNo {
        constraint exclusive;
    }
}
```

See also
<a href="#">Schema &gt; Object types</a>
<a href="#">SDL &gt; Object types</a>
<a href="#">DDL &gt; Object types</a>
<a href="#">Introspection &gt; Object types</a>

#### 4.4.8 Declaring functions

Define a function for counting reviews given a user name:

```
create function review_count(name: str) -> int64
using (
    with module default
    select count(
        (
            select Review
            filter .author.name = name
        )
    )
)
```

Drop a user-defined function:

```
drop function review_count(name: str);
```

Define and use polymorphic function:

```
db> create function make_name(name: str) -> str
... using ('my_name_ ' ++ name);
CREATE FUNCTION
db> create function make_name(name: int64) -> str
... using ('my_name_ ' ++ <str>name);
```

(continues on next page)

(continued from previous page)

**CREATE FUNCTION**

```
q> select make_name('Alice');
{'my_name_Alice'}
q> select make_name(42);
{'my_name_42'}
```

**See also**

<i>Schema &gt; Functions</i>
<i>SDL &gt; Functions</i>
<i>DDL &gt; Functions</i>
<i>Reference &gt; Function calls</i>
<i>Introspection &gt; Functions</i>
Tutorial > Advanced EdgeQL > User-Defined Functions

#### 4.4.9 Declaring aliases

Define an alias that merges some information from links as computed properties, this is a way of flattening a nested structure:

```
alias ReviewAlias := Review {
    # It will already have all the Review
    # properties and links.
    author_name := .author.name,
    movie_title := .movie.title,
}
```

Define an alias for traversing a *backlink*, this is especially useful for GraphQL access:

```
alias MovieAlias := Movie {
    # A computed link for accessing all the
    # reviews for this movie.
    reviews := .<movie[is Review]
```

**Note:** Aliases allow to use the full power of EdgeQL (expressions, aggregate functions, *backlink* navigation) from *GraphQL*.

The aliases defined above allow you to query `MovieAlias` with *GraphQL*.

<b>See also</b>
<i>Schema &gt; Aliases</i>
<i>SDL &gt; Aliases</i>
<i>DDL &gt; Aliases</i>

#### 4.4.10 Declaring annotations

Use annotations to add descriptions to types and links:

```
type Label {
    annotation description :=
        'Special label to stick on reviews';
    required property comments -> str;
    link review -> Review {
        annotation description :=
            'This review needs some attention';
    };
}
```

```
type Label {
    annotation description :=
        'Special label to stick on reviews';
    required comments: str;
    review: Review {
        annotation description :=
            'This review needs some attention';
    };
}
```

---

Retrieving the annotations can be done via an introspection query:

```
db> with module schema
... select ObjectType {
...     name,
...     annotations: {name, @value},
...     links: {name, annotations: {name, @value}}
... }
... filter .name = 'default::Label';
{
    Object {
        name: 'default::Label',
        annotations: {
            Object {
                name: 'std::description',
                @value: 'Special label to stick on reviews'
            }
        },
        links: {
            Object {
                name: 'review',
                annotations: {
                    Object {
                        name: 'std::description',
                        @value: 'Special label to stick on reviews'
                    }
                }
            },
        }
    }
},
```

(continues on next page)

(continued from previous page)

```

Object { name: '__type__', annotations: {} }
}
}
}

```

Alternatively, the annotations can be viewed by the following REPL command:

```

db> \d+ Label
type default::Label {
    annotation std::description := 'Special label to stick on reviews';
    required single link __type__ -> schema::Type {
        readonly := true;
    };
    single link review -> default::Review {
        annotation std::description := 'Special label to stick on reviews';
    };
    required single property comments -> std::str;
    required single property id -> std::uuid {
        readonly := true;
        constraint std::exclusive;
    };
};

```

See also
<a href="#">Schema &gt; Annotations</a>
<a href="#">SDL &gt; Annotations</a>
<a href="#">DDL &gt; Annotations</a>
<a href="#">Introspection &gt; Object types</a>

#### 4.4.11 Using the CLI

To initialize a new project:

```
$ edgedb project init
```

If an `edgedb.toml` file exists in the current directory, it will initialize a new project according to the settings defined in it.

Otherwise, a new project will be initialized and an `edgedb.toml` file and `dbschema` directory will be generated. For details on using projects, see the [dedicated guide](#).

Once initialized, you can run the CLI commands below without additional connection options. If you don't set up a project, you'll need to use `flags` to specify the target instance for each command.

Explicitly create a new EdgeDB instance `my_instance`:

```
$ edgedb instance create my_instance
```

Create a database:

```
$ edgedb database create special_db
OK: CREATE
```

Configure passwordless access (such as to a local development database):

```
$ edgedb configure insert Auth \
> --comment 'passwordless access' \
> --priority 1 \
> --method Trust
OK: CONFIGURE INSTANCE
```

Configure access that checks password (with a higher priority):

```
$ edgedb configure insert Auth \
> --comment 'password is required' \
> --priority 0 \
> --method SCRAM
OK: CONFIGURE INSTANCE
```

Connect to the default project database:

```
$ edgedb
EdgeDB 1.0-beta.2+ga7130d5c7.cv202104290000 (repl 1.0.0-beta.2)
Type \help for help, \quit to quit.
edgedb>
```

Connect to some specific database:

```
$ edgedb -d special_db
EdgeDB 1.0-beta.2+ga7130d5c7.cv202104290000 (repl 1.0.0-beta.2)
Type \help for help, \quit to quit.
special_db>
```

#### 4.4.12 Using the REPL

Execute a query. To execute a query in the REPL, terminate the statement with a semicolon and press “ENTER”.

```
db> select 5;
{5}
```

Alternatively, you can run the query without a semicolon by hitting Alt-Enter on Windows/Linux, or Esc+Return on macOS.

```
db> select 5
{5}
```

Type Alt+Enter to run the query without having the cursor to the end of the query.

---

**Note:** This doesn't work by default on macOS, however it's possible to enable it with a quick fix.

- in Terminal.app: Settings → Profiles → Keyboard → Check “Use Option as Meta key”
- in iTerm: Settings → Profiles → Keys → Let Option key: ESC+

Alternatively you can use the `esc+return` shortcut.

---



---

Use query parameters. If your query contains a parameter, you will be prompted for a value.

```
db> select 5 + <int64>$num;
Parameter <int64>$num: 6
{11}
```

---

#### 4.4.12.1 Commands

Options	<code>-v</code> = verbose <code>-s</code> = show system objects <code>-I</code> = case-sensitive match
<code>\d [-v] NAME</code>	Describe a schema object.
<code>\ds, \describe schema</code>	Describe the entire schema.
<code>\list databases</code> alias: <code>\l</code>	List databases.
<code>\list scalars [-sI] [pattern]</code> alias: <code>\ls</code>	List scalar types.
<code>\list types [-sI] [pattern]</code> alias: <code>\lt</code>	List object types.
<code>\list roles [-I]</code> alias: <code>\lr</code>	List roles.
<code>\list modules [-I]</code> alias: <code>\lm</code>	List modules.
<code>\list aliases [-Isv] [pattern]</code> alias: <code>\la</code>	List expression aliases.
<code>\list casts [-I] [pattern]</code> alias: <code>\lc</code>	List casts.
<code>\list indexes [-Isv] [pattern]</code> alias: <code>\li</code>	List indexes.
<code>\dump &lt;filename&gt;</code>	Dump the current database to file.
<code>\restore &lt;filename&gt;</code>	Restore the database from a dump file.
<code>\s, \history</code>	Show query history
<code>\e, \edit [N]</code>	Spawn \$EDITOR to edit history entry N. Then use the output as the input.
<code>\set [&lt;option&gt; [&lt;value&gt;]]</code>	View/change a setting. Type <code>\set</code> to see all available settings.
<code>\c, \connect [&lt;dbname&gt;]</code>	Connect to a particular database.

#### 4.4.12.2 Sample usage

List databases:

```
db> \ls
List of databases:
  db
  tutorial
```

---

Connect to a database:

```
db> \c my_new_project
my_new_project>
```

---

Describe an object type:

```
db> \d object Object
abstract type std::Object extending std::BaseObject {
    required single link __type__ -> schema::Type {
        readonly := true;
    };
    required single property id -> std::uuid {
        readonly := true;
    };
};
```

---

Describe a scalar type:

```
db> \d object decimal
scalar type std::decimal extending std::anynumeric;
```

---

Describe a function:

```
db> \d object sum
function std::sum(s: set of std::bigint) -> std::bigint {
    volatility := 'Immutable';
    annotation std::description := 'Return the sum of the set of numbers.';
    using sql function 'sum'
;};
function std::sum(s: set of std::int32) -> std::int64 {
    volatility := 'Immutable';
    annotation std::description := 'Return the sum of the set of numbers.';
    using sql function 'sum'
;};
function std::sum(s: set of std::decimal) -> std::decimal {
    volatility := 'Immutable';
    annotation std::description := 'Return the sum of the set of numbers.';
    using sql function 'sum'
```

(continues on next page)

(continued from previous page)

```
;};  
function std::sum(s: set of std::float32) -> std::float32 {  
    volatility := 'Immutable';  
    annotation std::description := 'Return the sum of the set of numbers.';  
    using sql function 'sum'  
};;  
function std::sum(s: set of std::int64) -> std::int64 {  
    volatility := 'Immutable';  
    annotation std::description := 'Return the sum of the set of numbers.';  
    using sql function 'sum'  
};;  
function std::sum(s: set of std::float64) -> std::float64 {  
    volatility := 'Immutable';  
    annotation std::description := 'Return the sum of the set of numbers.';  
    using sql function 'sum'  
};;
```

#### 4.4.13 Administering an instance

Create a database:

```
db> create database my_new_project;  
OK: CREATE DATABASE
```

Create a role:

```
db> create superuser role project;  
OK: CREATE ROLE
```

Configure passwordless access (such as to a local development database):

```
db> configure instance insert Auth {  
...     # Human-oriented comment helps figuring out  
...     # what authentication methods have been setup  
...     # and makes it easier to identify them.  
...     comment := 'passwordless access',  
...     priority := 1,  
...     method := (insert Trust),  
... };  
OK: CONFIGURE INSTANCE
```

Set a password for a role:

```
db> alter role project  
...     set password := 'super-password';  
OK: ALTER ROLE
```

---

Configure access that checks password (with a higher priority):

```
db> configure instance insert Auth {  
...     comment := 'password is required',  
...     priority := 0,  
...     method := (insert SCRAM),  
... };  
OK: CONFIGURE INSTANCE
```

---

Remove a specific authentication method:

```
db> configure instance reset Auth  
... filter .comment = 'password is required';  
OK: CONFIGURE INSTANCE
```

---

Run a script from command line:

```
cat myscript.edgeql | edgedb [<connection-option>...]
```

## 4.5 Contributing

### **edb-alt-title** Contributing to EdgeDB

EdgeDB is an open-source project, and we welcome contributions from our community. You can contribute by writing code or by helping us improve our documentation.

#### 4.5.1 Code

##### **edb-alt-title** Developing EdgeDB

This section describes how to build EdgeDB locally, how to use its internal tools, and how to contribute to it.

**Warning:** Code-changing pull requests without adding new tests might take longer time to be reviewed and merged.

##### 4.5.1.1 Building Locally

The following instructions should be used to create a “dev” build on Linux or macOS. Windows is not currently supported.

## Build Requirements

- GNU make version 3.80 or newer;
- C compiler (GCC or clang);
- Rust compiler and Cargo 1.65 or later;
- autotools;
- Python 3.10 dev package;
- Bison 1.875 or later;
- Flex 2.5.31 or later;
- Perl 5.8.3 or later;
- Zlib (zlib1-dev on Ubuntu);
- Readline dev package;
- Libuuid dev package;
- Node.js 14 or later;
- Yarn 1

On Ubuntu 22.10, these can be installed by running:

```
$ apt install make gcc rust-all autotools-dev python3.11-dev \
  python3.11-venv bison flex libreadline-dev perl zlib1g-dev \
  uuid-dev nodejs npm
$ npm i -g corepack
$ corepack enable && corepack prepare yarn@stable --activate
```

## Instructions

The easiest way to set up a development environment is to create a Python “venv” with all dependencies and commands installed into it.

1. Make a new directory that will contain checkouts of `edgedb` and `edgedb-python`. The name of the directory is arbitrary, we will use “dev” in this guide:

```
$ mkdir ~/dev
$ cd ~/dev
```

2. Clone the `edgedb` repository using `--recursive` to clone all submodules:

```
$ git clone --recursive https://github.com/edgedb/edgedb.git
```

3. Create a Python 3.10 virtual environment and activate it:

```
$ python3.10 -m venv edgedb-dev
$ source edgedb-dev/bin/activate
```

4. Build `edgedb` (the build will take a while):

```
$ cd edgedb
$ pip install -v -e "[test]"
```

In addition to compiling EdgeDB and all dependencies, this will also install the `edb` and `edgedb` command line tools into the current Python virtual environment.

It will also install libraries used during development.

5. Run tests:

```
$ edb test
```

The new virtual environment is now ready for development and can be activated at any time.

#### 4.5.1.2 Running Tests

To run all EdgeDB tests simply use the `$ edb test` command without arguments.

The command also supports running a few selected tests. To run all tests in a test case file:

```
$ edb test tests/test_edgeql_calls.py  
  
# or run two files:  
$ edb test tests/test_edgeql_calls.py tests/test_edgeql_for.py
```

To pattern-match a test by its name:

```
$ edb test -k test_edgeql_calls_01  
  
# or run all tests that contain "test_edgeql_calls":  
$ edb test -k test_edgeql_calls
```

See `$ edb test --help` for more options.

#### 4.5.1.3 Dev Server

Use the `$ edb server` command to start the development server.

You can then use another terminal to open a REPL to the server using the `$ edgedb` command, or connect to it using one of the language bindings.

#### 4.5.1.4 Test Databases

Use the `$ edb initfdb` command to create and populate databases that are used by unit tests.

### 4.5.2 Documentation

#### **edb-alt-title** Writing EdgeDB Documentation

We pride ourselves on having some of the best documentation around, but we want you to help us make it even better. Documentation is a great way to get started contributing to EdgeDB. Improvements to our documentation create a better experience for every developer coming through the door behind you.

Follow our general and style guidelines to make for a smooth contributing experience, both for us and for you. You may notice that the existing documentation doesn't always follow the guidelines laid out here. They are aspirational, so there are times when we intentionally break with them. Other times, bits of documentation may not be touched for a while and so may not reflect our current guidelines. These are great “low-hanging fruit” opportunities for your contributions!

#### 4.5.2.1 Guidelines

- **Avoid changes that don't fix an obvious mistake or add clarity.** This is subjective, but try to look at your changes with a critical eye. Do they fix errors in the original like misspellings or typos? Do they make existing prose more clear or accessible while maintaining accuracy? If you answered “yes” to either of those questions, this might be a great addition to our docs! If not, consider starting a discussion instead to see if your changes might be the exception to this guideline before submitting.
- **Keep commits and pull requests small.** We get it. It's more convenient to throw all your changes into a single pull request or even into a single commit. The problem is that, if some of the changes are good and others don't quite work, having everything in one bucket makes it harder to filter out the great changes from those that need more work.
- **Make spelling and grammar fixes in a separate pull request from any content changes.** These changes are quick to check and important to anyone reading the docs. We want to make sure they hit the live documentation as quickly as possible without being bogged down by other changes that require more intensive review.

#### 4.5.2.2 Style

- **Lines should be no longer than 79 characters.** This is enforced by linters as part of our CI process. Linting *can be disabled*, but this should not be used unless it's necessary and only for as long as it is necessary.
- **Remove trailing whitespace or whitespace on empty lines.**
- **Surround references to parameter named with asterisks.** You may be tempted to surround parameter names with double backticks (` `param``). We avoid that in favor of \*param\*, in order to distinguish between parameter references and inline code (which *should* be surrounded by double backticks).
- **EdgeDB is singular.** Choose “EdgeDB is” over “EdgeDB are” and “EdgeDB does” over “EdgeDB do.”
- **Use American English spellings.** Choose “color” over “colour” and “organize” over “organise.”
- **Use the Oxford comma.** When delineating a series, place a comma between each item in the series, even the one with the conjunction. Use “eggs, bacon, and juice” rather than “eggs, bacon and juice.”
- **Write in the simplest prose that is still accurate and expresses everything you need to convey.** You may be tempted to write documentation that sounds like a computer science textbook. Sometimes that's necessary, but in most cases, it isn't. Prioritize accuracy first and accessibility a close second.
- **Be careful using words that have a special meaning in the context of EdgeDB.** In casual speech or writing, you might talk about a “set” of something in a generic sense. Using the word this way in EdgeDB documentation might easily be interpreted as a reference to EdgeDB's *sets*. Avoid this kind of casual usage of key terms.

#### 4.5.2.3 Where to Find It

Most of our documentation (including this guide) lives in the `edgedb` repository in the `docs` directory.

Documentation for some of our client libraries lives inside the client's repo. If you don't find it in the `edgedb` repo at `docs/clients`, you'll probably find it alongside the client itself. These clients will also have documentation stubs inside the `edgedb` repository directing you to the documentation's location.

The [EdgeDB tutorial](#) is part of our web site repository. You'll find it in the `tutorial` directory.

Finally, our book for beginners titled [Easy EdgeDB](#) lives in its own repo.

#### 4.5.2.4 How to Build It

##### 4.5.2.4.1 edgedb/edgedb

The `edgedb` repository contains all of its documentation in the `docs/` directory. Run `make docs` to build the documentation in the `edgedb` repo. The repository contains a `Makefile` for all of Sphinx's necessary build options. The documentation will be built to `docs/build`.

To run tests, first *build EdgeDB locally*. Then run `edb test -k doc`.

Building the docs from this repo will not give you a high-fidelity representation of what users will see on the web site. For that, you may want to do a full documentation build.

##### 4.5.2.4.2 Full Documentation Build

A full documentation build requires more setup, but it is the only way to see your documentation exactly as the user will see it. This is not required, but it can be useful to help us review and approve your request more quickly by avoiding mistakes that would be easier to spot when they are fully rendered.

To build, clone [our website repository](#) and follow the installation instructions. Then run `yarn dev` to start a development server which also triggers a build of all the documentation.

---

**Note:** The watch task builds documentation changes, but it cannot trigger auto-reload in the browser. You will need to manually reload the browser to see changes made to the documentation.

---

#### 4.5.2.5 Sphinx and reStructuredText

Our documentation is first built through [Sphinx](#) and is written in [reStructuredText](#). If you're unfamiliar with `reStructuredText`, [the official primer](#) is a good place to start. [The official cheatsheet](#) serves as a great companion reference while you write. Sphinx also offers their own `reStructuredText` primer.

Sphinx not only builds the documentation but also extends `reStructuredText` to allow for a more ergonomic experience when crafting docs.

`ReStructuredText` is an easy-to-learn markup language built for documentation. Here are the most commonly used elements across our documentation.

##### 4.5.2.5.1 reStructuredText Basics

###### Headings

`ReStructuredText` headings are underlined (and sometimes overlined) with various characters. It's flexible about which characters map to which heading levels and will automatically assign heading levels to characters based on the hierarchy of the document.

To make it easier to quickly discern the level of a heading across our documentation, we use a consistent hierarchy across all pages.

1. = under and over- Used for the top-level heading which is usually the page title.
2. = under only
3. -

4. ^

### Example

```
=====
Page Title
=====

Section
=====

Sub-Section
-----

Sub-Sub-Section
^^^^^^^^^^^^^^^^^
```

If you need additional heading levels, you may use the `... rubric::` directive and pass it your heading by adding the heading text on the same line.

### Example

```
... rubric:: Yet Another Heading
```

## Inline Formatting

Text can be *italicized* by surrounding it with asterisks.

```
*italicized*
```

**Bold** text by surrounding it with double asterisks.

```
**Bold**
```

## Labels and Links

Labels make it easy to link across our documentation.

```
... _ref_eql_select_objects:
```

All pages must have a label at the top, but inner labels are added only when we need to link to them. Feel free to add a label to a section you need to link to. Follow the convention of `_ref_<main-section>_<page>_<section>` when naming labels. Check the page's main label at the top if you're not sure how to name your label. Append an underscore and the name of the section to the page's label. If you create a page, make sure you add a main label to the top of it.

Create internal links using the `:ref:` role. First find the label you want to link to. Reference the label's name in your role inside backticks (\`) removing the leading underscore as in the example below.

### Example

```
:ref:`ref_eql_select_objects`
```

### Rendered

*Selecting objects*

The label being linked can be on the same page as the link or on an entirely different page. Sphinx will find the label and link to the appropriate page and section.

You may also customize the link text.

#### Example

```
:ref:`our documentation on selecting objects <ref.eql_select_objects>`
```

#### Rendered

*our documentation on selecting objects*

To link to documentation for EdgeQL functions, statements, types, operators, or keywords, see the instructions in [Documenting EdgeQL](#).

## Special Paragraphs

Call out a paragraph as a note or warning using the appropriate directives.

#### Example

```
.. note::  
  
    This paragraph is a note.
```

#### Rendered

---

**Note:** This paragraph is a note.

---

#### Example

```
.. warning::  
  
    This paragraph is a warning.
```

#### Rendered

---

**Warning:** This paragraph is a warning.

---

You may also add a title to any of these paragraphs by passing it to the directive by placing it on the same line.

#### Example

```
.. note:: A Note  
  
    This paragraph is a note.
```

#### Rendered

---

**Note:** A Note

---

This paragraph is a note.

---

## Reusing Documentation

If you have documentation that will be reused in multiple contexts, you can write it in a separate `.rst` file and include that file everywhere it should appear.

```
.. include:: ../stdlib/constraint_table.rst
```

## Tables and Lists

We use tables and lists in a few different contexts.

### Example

```
.. list-table::  
  
    * - Arrays  
      - ``array<str>``  
    * - Tuples (unnamed)  
      - ``tuple<str, int64, bool>``  
    * - Tuples (named)  
      - ``tuple<name: str, age: int64, is_awesome: bool>``  
    * - Ranges  
      - ``range<float64>``
```

### Rendered

Arrays	<code>array&lt;str&gt;</code>
Tuples (unnamed)	<code>tuple&lt;str, int64, bool&gt;</code>
Tuples (named)	<code>tuple&lt;name: str, age: int64, is_awesome: bool&gt;</code>
Ranges	<code>range&lt;float64&gt;</code>

### Example

```
.. list-table::  
    :class: seealso  
  
    * - **See also**  
    * - :ref:`Schema > Access policies <ref_datamodel_access_policies>`  
    * - :ref:`SDL > Access policies <ref_eql_sdl_access_policies>`
```

### Rendered

<b>See also</b>
<i>Schema &gt; Access policies</i>
<i>SDL &gt; Access policies</i>

---

**Note:** The `seealso` class adds a spacer above the table to push the table away from the main page content.

---

### Example

Syntax	Inferred type
:eql:code:`select 3;`	:eql:type:`int64`
:eql:code:`select 3.14;`	:eql:type:`float64`
:eql:code:`select 314e-2;`	:eql:type:`float64`
:eql:code:`select 42n;`	:eql:type:`bigint`
:eql:code:`select 42.0n;`	:eql:type:`decimal`
:eql:code:`select 42e+100n;`	:eql:type:`decimal`

**Rendered**

Syntax	Inferred type
select 3;	<i>int64</i>
select 3.14;	<i>float64</i>
select 314e-2;	<i>float64</i>
select 42n;	<i>bigint</i>
select 42.0n;	<i>decimal</i>
select 42e+100n;	<i>decimal</i>

**4.5.2.5.2 Sphinx Basics****Tables of Contents**

Sphinx requires that every page in the documentation be referenced from a table of contents. Use the `.. toctree::` directive to create a table of contents.

**Example**

```
.. toctree::
   :maxdepth: 3
   :hidden:

   code
   documentation
```

Most of our tables of contents use the roles you see in this example to set a maximum depth of 3 and to hide the table of contents. This is not required though if other options make sense in your context. Even though the tables are hidden, their content still gets rendered in the left sidebar navigation.

We generally use relative references in the `toctree` directive which reference the pages relative to the location of the page that contains the directive. The order of the references in the directive determines their order in the sidebar navigation.

If any document is not included in any `toctree`, it will cause Sphinx to error on the build unless you add the `:orphan:` role to the top of the page. We don't want to use this technique for most pages although there are exceptions.

#### 4.5.2.6 Rendering Code

Use these tools to render code in your documentation contribution.

##### 4.5.2.6.1 Inline Code

Render inline code by surrounding it with double backticks:

###### Example

```
With the help of a ``with`` block, we can add filters, ordering, and
pagination clauses.
```

###### Rendered

With the help of a `with` block, we can add filters, ordering, and pagination clauses.

**Warning:** Marking up inline code with single backticks a la Markdown will throw an error in Sphinx when building the documentation.

##### 4.5.2.6.2 Code Blocks

```
.. code-block:: [<language>]

<code goes here>
```

Render a block of code. You can optionally provide a language argument. Below are the most common languages used in our docs:

- `bash`- Include the prompt and optionally the output. When a user clicks the “copy” button to copy the code, it will copy only the input without the prompt and output.

###### Example

```
.. code-block:: bash

$ edgedb configure set listen_addresses 127.0.0.1 ::1
```

###### Rendered

```
$ edgedb configure set listen_addresses 127.0.0.1 ::1
```

- `edgeql`- Used for queries.

###### Example

```
.. code-block:: edgeql

select BlogPost filter .id = <uuid>$blog_id;
```

###### Rendered

```
select BlogPost filter .id = <uuid>$blog_id;
```

- `edgeql-repl`- An alternative to vanilla `edgeql`. Include the prompt and optionally the output. When a user clicks the “copy” button to copy the code, it will copy only the input without the prompt and output.

**Example**

```
.. code-block:: edgeql-repl

db> insert Person { name := <str>$name };
Parameter <str>$name: Pat
{default::Person {id: e9009b00-8d4e-11ed-a556-c7b5bdd6cf7a}}
```

**Rendered**

```
db> insert Person { name := <str>$name };
Parameter <str>$name: Pat
{default::Person {id: e9009b00-8d4e-11ed-a556-c7b5bdd6cf7a}}
```

- go
- javascript
- python

**Example**

```
.. code-block:: javascript

await client.query("select 'I ' ++ <str>$name ++ '!';", {
    name: "rock and roll"
});
```

**Rendered**

```
await client.query("select 'I ' ++ <str>$name ++ '!';", {
    name: "rock and roll"
});
```

- `sdl`- Used for defining schema.

**Example**

```
.. code-block:: sdl

module default {
    type Person {
        required property name -> str { constraint exclusive };
    }
}
```

**Rendered**

```
module default {
    type Person {
        required property name -> str { constraint exclusive };
    }
}
```

- <language>-diff- Shows changes in a code block. Each line of code in these blocks must be prefixed by a character: + for an added line, - for a removed line, or an empty space for an unchanged line.

**Example**

```
.. code-block:: sdl-diff

    type Movie {
        -   property title -> str;
        +   required property title -> str;
            multi link actors -> Person;
    }
```

**Rendered**

```
type Movie {
    -   property title -> str;
    +   required property title -> str;
        multi link actors -> Person;
}
```

- No language- Formats the text as a code block but without syntax highlighting. Use this for syntaxes that do not offer highlighting or in cases where highlighting is unnecessary.

**Example**

```
.. code-block::

[{"id": "ea7bad4c-35d6-11ec-9519-0361f8abd380"}, {"id": "6ddbb04a-3c23-11ec-b81f-7b7516f2a868"}, {"id": "b233ca98-3c23-11ec-b81f-6ba8c4f0084e"}, ]
```

**Rendered**

```
[{"id": "ea7bad4c-35d6-11ec-9519-0361f8abd380"}, {"id": "6ddbb04a-3c23-11ec-b81f-7b7516f2a868"}, {"id": "b233ca98-3c23-11ec-b81f-6ba8c4f0084e"}, ]
```

---

**Note:** Code blocks without a language specified do not have a “copy” button.

---

#### 4.5.2.6.3 Code Tabs

```
.. tabs::
```

Tabs are used to present code examples in multiple languages. This can be useful when you want to show a query in, for example, both EdgeQL and the TypeScript query builder.

##### Example

```
.. tabs::

    .. code-tab:: edgeql

        insert Movie {
            title := 'Doctor Strange 2',
            release_year := 2022
        };

    .. code-tab:: typescript

        const query = e.insert(e.Movie, {
            title: 'Doctor Strange 2',
            release_year: 2022
        });

        const result = await query.run(client);
```

##### Rendered

Listing 1: edgeql

```
insert Movie {
    title := 'Doctor Strange 2',
    release_year := 2022
};
```

Listing 2: typescript

```
const query = e.insert(e.Movie, {
    title: 'Doctor Strange 2',
    release_year: 2022
});

const result = await query.run(client);
```

### 4.5.2.7 Documenting EdgeQL

Tools to help document EdgeQL are in the :eql: domain.

#### 4.5.2.7.1 Functions

To document a function use a .. eql:function:: directive. Include these elements:

- Specify the full function signature with a fully qualified name on the same line as the directive.
- Add a description of each parameter using :param \$<name>: description:. \$<name> must match the name of the parameter in function's signature. If a parameter is positional rather than named, its number should be used instead (e.g. \$1).
- Add a type for each parameter using :paramtype \$<name>: <type>. For example: :paramtype \$<name>: int64 declares that the type of the \$<name> parameter is int64. If a parameter has more than one valid type, list them separated by “or” like this: :paramtype \$<name>: int64 or str.
- Document the return value of the function with :return: and :returntype:. :return: marks a description of the return value and :returntype: its type.
- Finish with a few descriptive paragraphs and code samples. The first paragraph must be a single sentence no longer than 79 characters describing the function.

#### Example

```
.. eql:function:: std::array_agg(set of any, $a: any) -> array<any>

:param $1: input set
:paramtype $1: set of any

:param $a: description of this param
:paramtype $a: int64 or str

:return: array made of input set elements
:returntype: array<any>

Return the array made from all of the input set elements.

The ordering of the input set will be preserved if specified.
```

You can link to a function's documentation by using the :eql:func: role. For instance:

- :eql:func:`array\_agg`;
- :eql:func:`std::array\_agg`;

These will link to a function using the function's name as you have written in between the backticks followed by parentheses. Here are the above links rendered:

- `array_agg()`;
- `std::array_agg()`;

You can customize a link's label with this syntax: `:eql:func:`aggregate a set as an array <array_agg>``. Here's the rendered output: `aggregate a set as an array`

#### 4.5.2.7.2 Operators

Use the `.. eql:operator::` directive to document an operator. On the same line as the directive, provide a string argument of the format `<operator-id>: <operator-signature>`

Add a `:optype <operand-name>: <type>` field for each of the operator signature's operands to declare their types.

##### Example

```
.. eql:operator:: PLUS: A + B

:optype A: int64 or str or bytes
:optype B: any
:resulttype: any

Arithmetic addition.
```

You can link to an operator's documentation by using the `:eql:op:` role, followed by the operator's ID you specified in your argument to `.. eql:operator::`. For instance: `:eql:op:`plus`` which renders as `plus`. You can customize the link label like this: `:eql:op:`+ <plus>``, which renders as `+`.

#### 4.5.2.7.3 Statements

Use the `:eql-statement:` field to sections that describe a statement. Add the `:eql-haswith:` field if the statement supports a `with` block.

```
Select
=====

:eql-statement:
:eql-haswith:

``select``--retrieve or compute a set of values.

.. eql:synopsis::

    [ with <with-item> [, ...] ]

    select <expr>

    [ filter <filter-expr> ]

    [ order by <order-expr> [direction] [then ...] ]
```

(continues on next page)

(continued from previous page)

```
[ offset <offset-expr> ]
[ limit  <limit-expr> ] ;
```

After laying out the formal syntax, describe the function of each clause with a synopsis like this:

```
:eql:synopsis:`filter <filter-expr>`  
The optional ``filter`` clause, where :eql:synopsis:`<filter-expr>`  
is any expression that has a result of type :eql:type:`bool`.  
The condition is evaluated for every element in the set produced by  
the ``select`` clause. The result of the evaluation of the  
``filter`` clause is a set of boolean values. If at least one value  
in this set is ``true``, the input element is included, otherwise  
it is eliminated from the output.
```

These descriptions can each contain as many paragraphs as needed to adequately describe the clause. Follow the format used in the PostgreSQL documentation. See [the PostgreSQL SELECT statement reference page](#) for an example.

Use :eql:stmt:`select` to link to the statement's documentation. When rendered the link looks like this: [select](#). Customize the label with :eql:stmt:`the select statement <select>` which renders as this: [the select statement](#).

#### 4.5.2.7.4 Types

To document a type, use the .. eql:type:: directive. Follow the directive with the fully-qualified name of the type on the same line. The block should contain the type's description.

```
.. eql:type:: std::bytes  
A sequence of bytes.
```

To link to a type's documentation, use :eql:type:`bytes` which renders as [bytes](#). You may use the fully qualified name in your reference — :eql:type:`std::bytes` — which renders as [std::bytes](#). Both forms reference the same location in the documentation. Link labels can be customized with :eql:type:`the bytes type <bytes>` which renders like this: [the bytes type](#).

#### 4.5.2.7.5 Keywords

Document a keyword using the .. eql:keyword:: directive.

```
.. eql:keyword:: with  
The ``with`` block in EdgeQL is used to define aliases.
```

If a keyword is compound use a hyphen between each word.

```
.. eql:keyword:: set-of
```

To link to a keyword's documentation, use the :eql:kw: role like this: :eql:kw:`detached` which renders as [detached](#). You can customize the link label like this: :eql:kw:`the "detached" keyword <detached>` which renders as [the "detached" keyword](#).

#### 4.5.2.8 Documenting the EdgeQL CLI

Document a CLI command using the `cli:synopsis` directive like this:

##### Example

```
.. cli:synopsis::

    edgedb dump [<options>] <path>
```

##### Rendered

```
edgedb dump [<options>] <path>
```

The synopsis should follow the format used in the PostgreSQL documentation. See [the PostgreSQL SELECT statement reference page](#) for an example.

You can then document arguments and options using the `:cli:synopsis:` role.

##### Example

```
:cli:synopsis:`<path>`
    The name of the file to backup the database into.
```

##### Rendered

`<path>` The name of the file to backup the database into.

#### 4.5.2.9 Documentation Versioning

Since EdgeDB functionality is mostly consistent across versions, we offer a simple method of versioning documentation using two directives.

**Warning:** Although these are directives included in Sphinx, we have customized them to behave differently. Please read this documentation even if you're already familiar with the Sphinx directives mentioned here.

##### 4.5.2.9.1 New in Version

Content addressing anything new in a given version are marked with the `versionadded` directive. Provide the applicable version as an argument by placing it just after the directive on the same line.

The directive behaves differently depending on the context.

- When the directive has content (i.e., an indented paragraphs below the directive), that content will be shown or hidden based on the version switch.
- When the directive is placed immediately after a section header or inside a description block for a function, type, operator, statement, or keyword, that entire section or block is marked to be shown or hidden based on the version selected.
- When the directive is placed on the top line of any page before any content or reStructuredText labels (e.g., `.. _ref_eql_select:`), it applies to the entire page.

##### Example with Content

```
.. versionadded:: 2.0
```

This **is** a new feature that was added **in** EdgeDB 2.0.

### Rendered

This is a new feature that was added in EdgeDB 2.0.

---

**Note:** Change the version in the version selector dropdown to see how the rendered example changes.

---

### Section Example

```
Source deletion
^^^^^^^^^^^^^^^^^
```

```
.. versionadded:: 2.0
```

Source deletion policies determine what action should be taken when the `*source*` of a given link is deleted. They are declared with the `on source delete` clause.

```
...
```

### Rendered

See [the “Source deletion” section of the “Links” documentation](#) for a rendered section example of `.. versionadded:: 2.0`.

### Description Block Example

```
.. eql:type:: cal::date_duration
```

```
.. versionadded:: 2.0
```

A **type for** representing a span of time **in** days.

### Rendered

See [`cal::date\_duration`](#) for a rendered description block example of `.. versionadded:: 2.0`.

### Full-Page Example

```
.. versionadded:: 2.0
```

```
.. _ref_datamodel_globals:
```

```
=====
```

**Globals**

```
=====
```

```
...
```

### Rendered

See [the “Globals” documentation page](#) for a full-page example of `.. versionadded:: 2.0`.

### 4.5.2.9.2 Changed in Version

Use the `versionchanged` directive to mark content related to a change in existing functionality across EdgeDB versions. Provide the applicable version as an argument by placing it just after the directive on the same line.

Unlike `versionadded`, `versionchanged` is always used with content to show or hide that content based on the user's selection in the version dropdown.

#### Example

```
.. versionchanged:: 3.0

Starting with the upcoming EdgeDB 3.0, access policy restrictions will
not apply to any access policy expression. This means that when
reasoning about access policies it is no longer necessary to take other
policies into account. Instead, all data is visible for the purpose of
defining an access policy.
```

#### Rendered

Starting with the upcoming EdgeDB 3.0, access policy restrictions will **not** apply to any access policy expression. This means that when reasoning about access policies it is no longer necessary to take other policies into account. Instead, all data is visible for the purpose of *defining* an access policy.

---

**Note:** Change the version in the version selector dropdown to see how the rendered example changes.

---

### 4.5.2.10 Other Useful Tricks

#### 4.5.2.10.1 Temporarily Disabling Linting

`.. lint-off` and `.. lint-on` toggle linting off or on. In general, linting should stay on except in cases where it's impossible to keep it on. This might be when code or a URL must exceed the maximum line length of 79 characters.

You would typically use this by toggling linting off with `.. lint-off` just before the offending block and back on with `.. lint-on` after the block.

#### Example

```
.. lint-off

.. code-block::

    GET http://localhost:<port>/db/edgedb/edgeql?query=insert%20Person%20%7B%20name%20%3A
    %3D%20%3Cstr%3E$name%20%7D%3B&variables=%7B%22name%22%3A%20%22Pat%22%7D

.. lint-on
```

---

**Note:** This is actually a comment our linter pays attention to rather than a directive. As a result, it does not end with a colon (:) like a directive would.

---

**Note:** This does not render any visible output.

---

#### 4.5.2.10.2 Embedding a YouTube Video

Embed only videos from the EdgeDB YouTube channel

```
.. lint-off

.. raw:: html

    <div style="position: relative; padding-bottom: 56.25%; height: 0; overflow: hidden; max-width: 100%; height: auto;">
        <iframe src="https://www.youtube.com/embed/OZ_UURzDkow" frameborder="0" allowfullscreen style="position: absolute; top: 0; left: 0; width: 100%; height: 100%;"></iframe>
    </div>

.. lint-on
```

#### 4.5.2.10.3 Displaying Illustrations

Using the `.. eql:section-intro-page::` directive, you can display one of several illustrations. Pass the name of the illustration to the directive by placing it after the directive on the same line.

##### Example

```
.. eql:section-intro-page:: edgeql
```

##### Rendered

See [the list of illustration names](#) and [view the images they map to](#).

To make sure the project can continue to improve quickly, we have a few guidelines designed to make it easier for your contributions to make it into the project. General guidelines are presented here. You will find guidelines relevant only to code or documentation in those sections of the guide.

These are guidelines rather than hard rules. If you want to submit a pull request that strays from these, it might be a good idea to start a discussion about it first. Otherwise, it's possible your pull request might not be merged.

### 4.5.3 General Guidelines

- **Avoid making pull requests that do not have an associated Github Issue.** This could be an already existing issue or one you create yourself when you discover the problem. This will allow the team to help you scope your solution, warn you of potential gotchas, or give you a heads-up on solutions that are likely not feasible. It's a good idea to mention in the issue that you'd like to contribute code to resolve the issue. **If you're fixing something trivial like a typo**, an associated issue isn't necessary.
- **Write good commit messages.** The subject of your commit message — that's the first line — should tell us *what* you did. The body of your message — that's the rest of it — should tell us *why* you did it (unless that's self-evident).

#### 4.5.4 Thank You!

Thank you for contributing to EdgeDB! We love our open source community and want to foster a healthy contributor ecosystem. We're happy to have you as a part of it.



**STANDARD LIBRARY**

## 5.1 Generic

**edb-alt-title** Generic Functions and Operators

<code>anytype = anytype</code>	C.compares two values for equality.
<code>anytype != anytype</code>	C.compares two values for inequality.
<code>anytype ?= anytype</code>	C.compares two (potentially empty) values for equality.
<code>anytype ?!= anytype</code>	C.compares two (potentially empty) values for inequality.
<code>anytype &lt; anytype</code>	Less than operator.
<code>anytype &gt; anytype</code>	Greater than operator.
<code>anytype &lt;= anytype</code>	Less or equal operator.
<code>anytype &gt;= anytype</code>	Greater or equal operator.
<code>len()</code>	Returns the number of elements of a given value.
<code>contains()</code>	Returns true if the given sub-value exists within the given value.
<code>find()</code>	Returns the index of a given sub-value in a given value.

**Note:** In EdgeQL, any value can be compared to another as long as their types are compatible.

**operator anytype = anytype -> bool**

Compares two values for equality.

```
db> select 3 = 3.0;
{true}
db> select 3 = 3.14;
{false}
db> select [1, 2] = [1, 2];
{true}
db> select (1, 2) = (x := 1, y := 2);
{true}
db> select (x := 1, y := 2) = (a := 1, b := 2);
{true}
db> select 'hello' = 'world';
{false}
```

**operator anytype != anytype -> bool**

Compares two values for inequality.

```
db> select 3 != 3.0;
{false}
db> select 3 != 3.14;
{true}
db> select [1, 2] != [2, 1];
{false}
db> select (1, 2) != (x := 1, y := 2);
{false}
db> select (x := 1, y := 2) != (a := 1, b := 2);
{false}
db> select 'hello' != 'world';
{true}
```

**operator optional anytype ?= optional anytype -> bool**

Compares two (potentially empty) values for equality.

This works the same as a regular `=` operator, but also allows comparing an empty `{}` set. Two empty sets are considered equal.

```
db> select {1} ?= {1.0};
{true}
db> select {1} ?= <int64>{};
{false}
db> select <int64>{} ?= <int64>{};
{true}
```

**operator optional anytype ?!= optional anytype -> bool**

Compares two (potentially empty) values for inequality.

This works the same as a regular `=` operator, but also allows comparing an empty `{}` set. Two empty sets are considered equal.

```
db> select {2} ?!= {2};
{false}
db> select {1} ?!= <int64>{};
{true}
db> select <bool>{} ?!= <bool>{};
{false}
```

**operator anytype < anytype -> bool**

Less than operator.

The operator returns `true` if the value of the left expression is less than the value of the right expression:

```
db> select 1 < 2;
{true}
db> select 2 < 2;
{false}
```

(continues on next page)

(continued from previous page)

```
db> select 'hello' < 'world';
{true}
db> select (1, 'hello') < (1, 'world');
{true}
```

**operator `anytype > anytype -> bool`**

Greater than operator.

The operator returns `true` if the value of the left expression is greater than the value of the right expression:

```
db> select 1 > 2;
{false}
db> select 3 > 2;
{true}
db> select 'hello' > 'world';
{false}
db> select (1, 'hello') > (1, 'world');
{false}
```

**operator `anytype <= anytype -> bool`**

Less or equal operator.

The operator returns `true` if the value of the left expression is less than or equal to the value of the right expression:

```
db> select 1 <= 2;
{true}
db> select 2 <= 2;
{true}
db> select 3 <= 2;
{false}
db> select 'hello' <= 'world';
{true}
db> select (1, 'hello') <= (1, 'world');
{true}
```

**operator `anytype >= anytype -> bool`**

Greater or equal operator.

The operator returns `true` if the value of the left expression is greater than or equal to the value of the right expression:

```
db> select 1 >= 2;
{false}
db> select 2 >= 2;
{true}
db> select 3 >= 2;
{true}
db> select 'hello' >= 'world';
```

(continues on next page)

(continued from previous page)

```
{false}
db> select (1, 'hello') >= (1, 'world');
{false}
```

```
function std::len → int64
function std::len → int64
function std::len → int64
```

**Index Keywords** length count array

Returns the number of elements of a given value.

This function works with the `str`, `bytes` and `array` types:

```
db> select len('foo');
{3}

db> select len(b'bar');
{3}

db> select len([2, 5, 7]);
{3}
```

```
function std::contains → bool
function std::contains → bool
function std::contains → bool
function std::contains → std::bool
function std::contains → std::bool
```

**Index Keywords** find strpos strstr position array

Returns true if the given sub-value exists within the given value.

When *haystack* is a `str` or a `bytes` value, this function will return `true` if it contains *needle* as a subsequence within it or `false` otherwise:

```
db> select contains('qwerty', 'we');
{true}

db> select contains(b'qwerty', b'42');
{false}
```

When *haystack* is an `array`, the function will return `true` if the array contains the element specified as *needle* or `false` otherwise:

```
db> select contains([2, 5, 7, 2, 100], 2);
{true}
```

When *haystack* is a `range`, the function will return `true` if it contains either the specified sub-range or element. The function will return `false` otherwise.

```
db> select contains(range(1, 10), range(2, 5));
{true}

db> select contains(range(1, 10), range(2, 15));
{false}

db> select contains(range(1, 10), 2);
{true}

db> select contains(range(1, 10), 10);
{false}
```

```
function std::find → int64
function std::find → int64
function std::find → int64
```

**Index Keywords** find strpos strstr position array

Returns the index of a given sub-value in a given value.

When *haystack* is a *str* or a *bytes* value, the function will return the index of the first occurrence of *needle* in it.

When *haystack* is an *array*, this will return the index of the the first occurrence of the element passed as *needle*. For *array* inputs it is also possible to provide an optional *from\_pos* argument to specify the position from which to start the search.

If the *needle* is not found, return -1.

```
db> select find('qwerty', 'we');
{1}

db> select find(b'qwerty', b'42');
{-1}

db> select find([2, 5, 7, 2, 100], 2);
{0}

db> select find([2, 5, 7, 2, 100], 2, 1);
{3}
```

## 5.2 Sets

**edb-alt-title** Set Functions and Operators

**index** set aggregate

<code>distinct set</code>	Produces a set of all unique elements in the given set.
<code>anytype in set</code>	Checks if a given element is a member of a given set.
<code>set union set</code>	Merges two sets.
<code>set intersect set</code>	Produces a set containing the common items between the given sets.
<code>set except set</code>	Produces a set of all items in the first set which are not in the second.
<code>exists set</code>	Determines whether a set is empty or not.
<code>set if bool else set</code>	Produces one of two possible results based on a given condition.
<code>optional anytype ?? set</code>	Produces the first of its operands that is not an empty set.
<code>detached</code>	Detaches the input set reference from the current scope.
<code>anytype [is type]</code>	Filters a set based on its type. Will return back the specified type.
<code>assert_distinct()</code>	Checks that the input set contains only unique elements.
<code>assert_single()</code>	Checks that the input set contains no more than one element.
<code>assert_exists()</code>	Checks that the input set contains at least one element.
<code>count()</code>	Returns the number of elements in a set.
<code>array_agg()</code>	Returns an array made from all of the input set elements.
<code>sum()</code>	Returns the sum of the set of numbers.
<code>all()</code>	Returns true if none of the values in the given set are false.
<code>any()</code>	Returns true if any of the values in the given set is true.
<code>enumerate()</code>	Returns a set of tuples in the form of (index, element).
<code>min()</code>	Returns the smallest value in the given set.
<code>max()</code>	Returns the largest value in the given set.
<code>math::mean()</code>	Returns the arithmetic mean of the input set.
<code>math::stddev()</code>	Returns the sample standard deviation of the input set.
<code>math::stddev_pop()</code>	Returns the population standard deviation of the input set.
<code>math::var()</code>	Returns the sample variance of the input set.
<code>math::var_pop()</code>	Returns the population variance of the input set.

**operator distinct set of anytype -> set of anytype**

Produces a set of all unique elements in the given set.

`distinct` is a set operator that returns a new set where no member is equal to any other member.

```
db> select distinct {1, 2, 2, 3};
{1, 2, 3}
```

**operator anytype in set of anytype -> bool****operator anytype not in set of anytype -> bool****Index Keywords** intersection

Checks if a given element is a member of a given set.

Set membership operators `in` and `not in` test whether each element of the left operand is present in the right operand. This means supplying a set as the left operand will produce a set of boolean results, one for each element in the left operand.

```
db> select 1 in {1, 3, 5};
{true}

db> select 'Alice' in User.name;
```

(continues on next page)

(continued from previous page)

```
{true}

db> select {1, 2} in {1, 3, 5};
{true, false}
```

This operator can also be used to implement set intersection:

```
db> with
...     A := {1, 2, 3, 4},
...     B := {2, 4, 6}
... select A filter A in B;
{2, 4}
```

#### **operator set of anytype union set of anytype -> set of anytype**

Merges two sets.

Since EdgeDB sets are formally multisets, `union` is a *multiset sum*, so effectively it merges two multisets keeping all of their members.

For example, applying `union` to `{1, 2, 2}` and `{2}`, results in `{1, 2, 2, 2}`.

If you need a distinct union, wrap it with the `distinct` operator.

#### **operator**

#### **set of anytype intersect set of anytype**

**-> set of anytype**

Produces a set containing the common items between the given sets.

---

**Note:** The ordering of the returned set may not match that of the operands.

---

If you need a distinct intersection, wrap it with the `distinct` operator.

#### **operator**

#### **set of anytype except set of anytype**

**-> set of anytype**

Produces a set of all items in the first set which are not in the second.

---

**Note:** The ordering of the returned set may not match that of the operands.

---

If you need a distinct set of exceptions, wrap it with the `distinct` operator.

#### **operator**

#### **set of anytype if bool else set of anytype**

**-> set of anytype**

**Index Keywords** if else ifelse elif ternary

Produces one of two possible results based on a given condition.

```
<left_expr> if <condition> else <right_expr>
```

If the <condition> is true, the if...else expression produces the value of the <left\_expr>. If the <condition> is false, however, the if...else expression produces the value of the <right\_expr>.

```
db> select 'hello' if 2 * 2 = 4 else 'bye';
{'hello'}
```

if..else expressions can be chained when checking multiple conditions is necessary:

```
db> with color := 'yellow'
... select 'Apple' if color = 'red' else
...      'Banana' if color = 'yellow' else
...      'Orange' if color = 'orange' else
...      'Other';
{'Banana'}
```

---

**operator optional `anytype` ?? `set of anytype`**

-> `set of anytype`

Produces the first of its operands that is not an empty set.

This evaluates to A for an non-empty A, otherwise evaluates to B.

A typical use case of the coalescing operator is to provide default values for optional properties:

```
# Get a set of tuples (<issue name>, <priority>)
# for all issues.
select (Issue.name, Issue.priority.name ?? 'n/a');
```

Without the coalescing operator, the above query will skip any Issue without priority.

---

**operator detached `set of anytype` -> `set of anytype`**

Detaches the input set reference from the current scope.

A detached expression allows referring to some set as if it were defined in the top-level with block. detached expressions ignore all current scopes in which they are nested. This makes it possible to write queries that reference the same set reference in multiple places.

```
update User
filter .name = 'Dave'
set {
    friends := (select detached User filter .name = 'Alice'),
    coworkers := (select detached User filter .name = 'Bob')
};
```

Without detached, the occurrences of User inside the set shape would be *bound* to the set of users named "Dave". However, in this context we want to run an unrelated query on the “unbound” User set.

```
# does not work!
update User
filter .name = 'Dave'
set {
    friends := (select User filter .name = 'Alice'),
    coworkers := (select User filter .name = 'Bob')
};
```

Instead of explicitly detaching a set, you can create a reference to it in a `with` block. All declarations inside a `with` block are implicitly detached.

```
with U1 := User,
      U2 := User
update User
filter .name = 'Dave'
set {
    friends := (select U1 filter .name = 'Alice'),
    coworkers := (select U2 filter .name = 'Bob')
};
```

#### operator exists set of anytype -> bool

Determines whether a set is empty or not.

`exists` is an aggregate operator that returns a singleton set `{true}` if the input set is not empty, and returns `{false}` otherwise:

```
db> select exists {1, 2};
{true}
```

#### operator anytype [is type] -> anytype

**Index Keywords** is type intersection

Filters a set based on its type. Will return back the specified type.

The type intersection operator removes all elements from the input set that aren't of the specified type. Additionally, since it guarantees the type of the result set, all the links and properties associated with the specified type can now be used on the resulting expression. This is especially useful in combination with [backlinks](#).

Consider the following types:

```
type User {
    required property name -> str;
}

abstract type Owned {
    required link owner -> User;
}

type Issue extending Owned {
    required property title -> str;
}

type Comment extending Owned {
    required property body -> str;
}
```

```
type User {
    required name: str;
}
```

(continues on next page)

(continued from previous page)

```
abstract type Owned {
    required owner: User;
}

type Issue extending Owned {
    required title: str;
}

type Comment extending Owned {
    required body: str;
}
```

The following expression will get all *Objects* owned by all users (if there are any):

```
select User.<owner;
```

By default, *backlinks* don't infer any type information beyond the fact that it's an *Object*. To ensure that this path specifically reaches *Issue*, the type intersection operator must then be used:

```
select User.<owner[is Issue];

# With the use of type intersection it's possible to refer to
# specific property of Issue now:
select User.<owner[is Issue].title;
```

**function std::assert\_distinct** → set of anytype

**Index Keywords** multiplicity uniqueness

Checks that the input set contains only unique elements.

If the input set contains duplicate elements (i.e. it is not a *proper set*), `assert_distinct` raises a `ConstraintViolationError`. Otherwise, this function returns the input set.

This function is useful as a runtime distinctness assertion in queries and computed expressions that should always return proper sets, but where static multiplicity inference is not capable enough or outright impossible. An optional *message* named argument can be used to customize the error message:

```
db> select assert_distinct(
...     (select User filter .groups.name = "Administrators")
...     union
...     (select User filter .groups.name = "Guests")
... )
{default::User {id: ...}};

db> select assert_distinct(
...     (select User filter .groups.name = "Users")
...     union
...     (select User filter .groups.name = "Guests")
... )
ERROR: ConstraintViolationError: assert_distinct violation: expression
       returned a set with duplicate elements.
```

(continues on next page)

(continued from previous page)

```
db> select assert_distinct(
...   (select User filter .groups.name = "Users")
...   union
...   (select User filter .groups.name = "Guests"),
...   message := "duplicate users!"
...
... )
ERROR: ConstraintViolationError: duplicate users!
```

**function std::assert\_single** → set of anytype**Index Keywords** cardinality singleton

Checks that the input set contains no more than one element.

If the input set contains more than one element, `assert_single` raises a `CardinalityViolationError`. Otherwise, this function returns the input set.This function is useful as a runtime cardinality assertion in queries and computed expressions that should always return sets with at most a single element, but where static cardinality inference is not capable enough or outright impossible. An optional `message` named argument can be used to customize the error message.

```
db> select assert_single((select User filter .name = "Unique"))
{default::User {id: ...} }

db> select assert_single((select User))
ERROR: CardinalityViolationError: assert_single violation: more than
      one element returned by an expression

db> select assert_single((select User), message := "too many users!")
ERROR: CardinalityViolationError: too many users!
```

**function std::assert\_exists** → set of anytype**Index Keywords** cardinality existence empty

Checks that the input set contains at least one element.

If the input set is empty, `assert_exists` raises a `CardinalityViolationError`. Otherwise, this function returns the input set.This function is useful as a runtime existence assertion in queries and computed expressions that should always return sets with at least a single element, but where static cardinality inference is not capable enough or outright impossible. An optional `message` named argument can be used to customize the error message.

```
db> select assert_exists((select User filter .name = "Administrator"))
{default::User {id: ...} }

db> select assert_exists((select User filter .name = "Nonexistent"))
ERROR: CardinalityViolationError: assert_exists violation: expression
      returned an empty set.

db> select assert_exists(
...   (select User filter .name = "Nonexistent"),
```

(continues on next page)

(continued from previous page)

```
...     message := "no users!"
...
ERROR: CardinalityViolationError: no users!
```

**function std::count** → int64**Index Keywords** aggregate

Returns the number of elements in a set.

```
db> select count({2, 3, 5});
{3}

db> select count(User); # number of User objects in db
{4}
```

```
function std::sum → int64
function std::sum → int64
function std::sum → float32
function std::sum → float64
function std::sum → bigint
function std::sum → decimal
```

**Index Keywords** aggregate

Returns the sum of the set of numbers.

The result type depends on the input set type. The general rule of thumb is that the type of the input set is preserved (as if a simple `+` was used) while trying to reduce the chance of an overflow (so all integers produce `int64` sum).

```
db> select sum({2, 3, 5});
{10}

db> select sum({0.2, 0.3, 0.5});
{1.0}
```

**function std::all** → bool**Index Keywords** aggregate

Returns `true` if none of the values in the given set are `false`.

The result is `true` if all of the *values* are `true` or the set of *values* is `{}`, with `false` returned otherwise.

```
db> select all(<bool>{});
{true}

db> select all({1, 2, 3, 4} < 4);
{false}
```

---

**function std::any** → bool

**Index Keywords** aggregate

Returns `true` if any of the values in the given set is `true`.

The result is `true` if any of the *values* are `true`, with `false` returned otherwise.

```
db> select any(<bool>{});  
{false}  
  
db> select any({1, 2, 3, 4} < 4);  
{true}
```

---

**function std::enumerate** → set of tuple<int64, anytype>

**Index Keywords** enumerate

Returns a set of tuples in the form of `(index, element)`.

The `enumerate()` function takes any set and produces a set of tuples containing the zero-based index number and the value for each element.

---

**Note:** The ordering of the returned set is not guaranteed, however, the assigned indexes are guaranteed to be in order of the original set.

---

```
db> select enumerate({2, 3, 5});  
{(1, 3), (0, 2), (2, 5)}
```

```
db> select enumerate(User.name);  
{(0, 'Alice'), (1, 'Bob'), (2, 'Dave')}
```

---

**function std::min** → optional anytype

**Index Keywords** aggregate

Returns the smallest value in the given set.

```
db> select min({-1, 100});  
{-1}
```

---

**function std::max** → optional anytype

**Index Keywords** aggregate

Returns the largest value in the given set.

```
db> select max({-1, 100});  
{100}
```

---

## 5.3 Types

### edb-alt-title Type Operators

<code>is type</code>	Type checking operator.
<code>type / type</code>	Type union operator.
<code>&lt;type&gt; val</code>	Type cast operator.
<code>typeof anytype</code>	Static type inference operator.
<code>introspect type</code>	Static type introspection operator.

---

```
operator anytype is type -> bool
operator anytype is not type -> bool
```

Type checking operator.

Check if A is an instance of B or any of B's subtypes.

Type-checking operators `is` and `is not` that test whether the left operand is of any of the types given by the comma-separated list of types provided as the right operand.

Note that B is special and is not any kind of expression, so it does not in any way participate in the interactions of sets and longest common prefix rules.

```
db> select 1 is int64;
{true}

db> select User is not SystemUser
... filter User.name = 'Alice';
{true}

db> select User is (Text | Named);
{true, ..., true} # one for every user instance
```

---

```
operator type | type -> type
```

**Index Keywords** poly polymorphism polymorphic queries nested shapes

Type union operator.

This operator is only valid in contexts where type checking is done. The most obvious use case is with the `is` and `is not`. The operator allows to refer to a union of types in order to check whether a value is of any of these types.

```
db> select User is (Text | Named);
{true, ..., true} # one for every user instance
```

It can similarly be used when specifying a link target type. The same logic then applies: in order to be a valid link target the object must satisfy `object is (A | B | C)`.

```
abstract type Named {
    required property name -> str;
}
```

(continues on next page)

(continued from previous page)

```

abstract type Text {
    required property body -> str;
}

type Item extending Named;

type Note extending Text;

type User extending Named {
    multi link stuff -> Named | Text;
}

```

```

abstract type Named {
    required name: str;
}

abstract type Text {
    required body: str;
}

type Item extending Named;

type Note extending Text;

type User extending Named {
    multi stuff: Named | Text;
}

```

With the above schema, the following would be valid:

```

db> insert Item {name := 'cube'};
{Object { id: <uuid>'...' }}
db> insert Note {body := 'some reminder'};
{Object { id: <uuid>'...' }}
db> insert User {
...     name := 'Alice',
...     stuff := Note, # all the notes
... };
{Object { id: <uuid>'...' }}
db> insert User {
...     name := 'Bob',
...     stuff := Item, # all the items
... };
{Object { id: <uuid>'...' }}
db> select User {
...     name,
...     stuff: {
...         [is Named].name,
...         [is Text].body
...     }
... };
{
```

(continues on next page)

(continued from previous page)

```
Object {
    name: 'Alice',
    stuff: {Object { name: {}, body: 'some reminder' }}
},
Object {
    name: 'Bob',
    stuff: {Object { name: 'cube', body: {} }}
}
```

**operator < type > anytype -> anytype**

Type cast operator.

A type cast operator converts the specified value to another value of the specified type:

`"<" <type> ">" <expression>`The `<type>` must be a valid *type expression* denoting a non-abstract scalar or a container type.

Type cast is a run-time operation. The cast will succeed only if a type conversion was defined for the type pair, and if the source value satisfies the requirements of a target type. EdgeDB allows casting any scalar.

It is illegal to cast one *Object* into another. The only way to construct a new *Object* is by using *insert*. However, the *type intersection* can be used to achieve an effect similar to casting for Objects.

When a cast is applied to an expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined.

Examples:

```
db> # cast a string literal into an integer
... select <int64>"42";
{42}

db> # cast an array of integers into an array of str
... select <array<str>>[1, 2, 3];
[['1', '2', '3']]

db> # cast an issue number into a string
... select <str>example::Issue.number;
{'142'}
```

Casts also work for converting tuples or declaring different tuple element names for convenience.

```
db> select <tuple<int64, str>>(1, 3);
{[1, '3']}

db> with
...     # a test tuple set, that could be a result of
...     # some other computation
...     stuff := (1, 'foo', 42)
... select (
...     # cast the tuple into something more convenient
```

(continues on next page)

(continued from previous page)

```
...     <tuple<a: int64, name: str, b: int64>>stuff
... ).name;  # access the 'name' element
{'foo'}
```

An important use of *casting* is in defining the type of an empty set {}, which can be required for purposes of type disambiguation.

```
with module example
select Text {
    name :=
        Text[is Issue].name IF Text is Issue ELSE
        <str>{},
        # the cast to str is necessary here, because
        # the type of the computed expression must be defined
    body,
};
```

Casting empty sets is also the only situation where casting into an *Object* is valid:

```
with module example
select User {
    name,
    friends := <User>{}
    # the cast is the only way to indicate that the
    # computed link 'friends' is supposed to refer to
    # a set of Users
};
```

For more information about casting between different types consult the *casting table*.

## operator `typeof` `anytype` -> `type`

**Index Keywords** type introspect introspection

Static type inference operator.

This operator converts an expression into a type, which can be used with *is*, *is not*, and *introspect*.

Currently, `typeof` operator only supports *scalars* and *objects*, but **not** the *collections* as a valid operand.

Consider the following types using links and properties with names that don't indicate their respective target types:

```
type Foo {
    property bar -> int16;
    link baz -> Bar;
}

type Bar extending Foo;
```

```
type Foo {
    bar: int16;
    baz: Bar;
```

(continues on next page)

(continued from previous page)

```

}

type Bar extending Foo;
```

We can use `typeof` to determine if certain expression has the same type as the property `bar`:

```

db> insert Foo { bar := 1 };
{Object { id: <uuid>'...' }}
db> select (Foo.bar / 2) is typeof Foo.bar;
{false}
```

To determine the actual resulting type of an expression we can use `introspect`:

```

db> select introspect (typeof Foo.bar).name;
{'std::int16'}
db> select introspect (typeof (Foo.bar / 2)).name;
{'std::float64'}
```

Similarly, we can use `typeof` to discriminate between the different `Foo` objects that can and cannot be targets of link `baz`:

```

db> insert Bar { bar := 2 };
{Object { id: <uuid>'...' }}
db> select Foo {
    ...     bar,
    ...     can_be_baz := Foo is typeof Foo.baz
    ... };
{
    Object { bar: 1, can_be_baz: false },
    Object { bar: 2, can_be_baz: true }
}
```

### **operator introspect type -> schema::Type**

**Index Keywords** type `typeof` `introspection`

Static type introspection operator.

This operator returns the `introspection type` corresponding to type provided as operand. It works well in combination with `typeof`.

Currently, the `introspect` operator only supports `scalar types` and `object types`, but **not** the `collection types` as a valid operand.

Consider the following types using links and properties with names that don't indicate their respective target types:

```

type Foo {
    property bar -> int16;
    link baz -> Bar;
}

type Bar extending Foo;
```

```
type Foo {
    bar: int16;
    baz: Bar;
}

type Bar extending Foo;
```

```
db> select (introspect int16).name;
{'std::int16'}
db> select (introspect Foo).name;
{'default::Foo'}
db> select (introspect typeof Foo.bar).name;
{'std::int16'}
```

**Note:** For any *object type* `SomeType` the expressions `introspect SomeType` and `introspect typeof SomeType` are equivalent as the object type name is syntactically identical to the *expression* denoting the set of those objects.

There's an important difference between the combination of `introspect typeof SomeType` and `SomeType.__type__` expressions when used with objects. `introspect typeof SomeType` is statically evaluated and does not take in consideration the actual objects contained in the `SomeType` set. Conversely, `SomeType.__type__` is the actual set of all the types reachable from all the `SomeType` objects. Due to inheritance statically inferred types and actual types may not be the same (although the actual types will always be a subtype of the statically inferred types):

```
db> # first let's make sure we don't have any Foo objects
... delete Foo;
{ there may be some deleted objects here }
db> select (introspect typeof Foo).name;
{'default::Foo'}
db> select Foo.__type__.name;
{}
db> # let's add an object of type Foo
... insert Foo;
{Object { id: <uuid>'...' }}
db> # Bar is also of type Foo
... insert Bar;
{Object { id: <uuid>'...' }}
db> select (introspect typeof Foo).name;
{'default::Foo'}
db> select Foo.__type__.name;
{'default::Bar', 'default::Foo'}
```

## 5.4 Math

**edb-alt-title** Mathematical Functions

<code>math::abs()</code>	Returns the absolute value of the input.
<code>math::ceil()</code>	Rounds up a given value to the nearest integer.
<code>math::floor()</code>	Rounds down a given value to the nearest integer.
<code>math::ln()</code>	Returns the natural logarithm of a given value.
<code>math::lg()</code>	Returns the base 10 logarithm of a given value.
<code>math::log()</code>	Returns the logarithm of a given value in the specified base.
<code>math::mean()</code>	Returns the arithmetic mean of the input set.
<code>math::stddev()</code>	Returns the sample standard deviation of the input set.
<code>math::stddev_pop()</code>	Returns the population standard deviation of the input set.
<code>math::var()</code>	Returns the sample variance of the input set.
<code>math::var_pop()</code>	Returns the population variance of the input set.

---

**function** `math::abs` → anyreal

**Index Keywords** absolute

Returns the absolute value of the input.

```
db> select math::abs(1);
{1}
db> select math::abs(-1);
{1}
```

---

**function** `math::ceil` → float64

**function** `math::ceil` → float64

**function** `math::ceil` → bigint

**function** `math::ceil` → decimal

**Index Keywords** round

Rounds up a given value to the nearest integer.

```
db> select math::ceil(1.1);
{2}
db> select math::ceil(-1.1);
{-1}
```

---

**function** `math::floor` → float64

**function** `math::floor` → float64

**function** `math::floor` → bigint

**function** `math::floor` → decimal

**Index Keywords** round

Rounds down a given value to the nearest integer.

```
db> select math::floor(1.1);
{1}
db> select math::floor(-1.1);
{-2}
```

**function math::ln** → float64  
**function math::ln** → float64  
**function math::ln** → decimal

**Index Keywords** logarithm

Returns the natural logarithm of a given value.

```
db> select 2.718281829 ^ math::ln(100);
{100.0000009164575}
```

**function math::lg** → float64  
**function math::lg** → float64  
**function math::lg** → decimal

**Index Keywords** logarithm

Returns the base 10 logarithm of a given value.

```
db> select 10 ^ math::lg(42);
{42.00000000000001}
```

**function math::log** → decimal

**Index Keywords** logarithm

Returns the logarithm of a given value in the specified base.

```
db> select 3 ^ math::log(15n, base := 3n);
{15.000000000000005n}
```

**function math::mean** → float64  
**function math::mean** → float64  
**function math::mean** → decimal

**Index Keywords** average avg

Returns the arithmetic mean of the input set.

```
db> select math::mean({1, 3, 5});
{3}
```

**function math::stddev** → float64  
**function math::stddev** → float64  
**function math::stddev** → decimal

**Index Keywords** average

Returns the sample standard deviation of the input set.

```
db> select math::stddev({1, 3, 5});  
{2}
```

```
function math::stddev_pop → float64  
function math::stddev_pop → float64  
function math::stddev_pop → decimal
```

**Index Keywords** average

Returns the population standard deviation of the input set.

```
db> select math::stddev_pop({1, 3, 5});  
{1.6329931618545}
```

---

```
function math::var → float64  
function math::var → float64  
function math::var → decimal
```

**Index Keywords** average

Returns the sample variance of the input set.

```
db> select math::var({1, 3, 5});  
{4}
```

---

```
function math::var_pop → float64  
function math::var_pop → float64  
function math::var_pop → decimal
```

**Index Keywords** average

Returns the population variance of the input set.

```
db> select math::var_pop({1, 3, 5});  
{2.6666666666667}
```

## 5.5 Strings

**edb-alt-title** String Functions and Operators

<code>str</code>	String
<code>str[i]</code>	String indexing.
<code>str[from:to]</code>	String slicing.
<code>str ++ str</code>	String concatenation.
<code>str like pattern</code>	Case-sensitive simple string matching.
<code>str ilike pattern</code>	Case-insensitive simple string matching.
<code>= != ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>to_str()</code>	Return string representation of the input value.
<code>len()</code>	Return string's length.
<code>contains()</code>	Test if a string contains a substring.
<code>find()</code>	Find index of a substring.
<code>str_lower()</code>	Return a lowercase copy of the input string.
<code>str_upper()</code>	Return an uppercase copy of the input string.
<code>str_title()</code>	Return a titlecase copy of the input string.
<code>str_pad_start()</code>	Return the input string padded at the start to the length n.
<code>str_pad_end()</code>	Return the input string padded at the end to the length n.
<code>str_trim()</code>	Return the input string with trim characters removed from both ends.
<code>str_trim_start()</code>	Return the input string with all trim characters removed from its start.
<code>str_trim_end()</code>	Return the input string with all trim characters removed from its end.
<code>str_repeat()</code>	Repeat the input string n times.
<code>str_replace()</code>	Replace all occurrences of old substring with the new one.
<code>str_reverse()</code>	Reverse the order of the characters in the string.
<code>str_split()</code>	Split a string into an array using a delimiter.
<code>re_match()</code>	Find the first regular expression match in a string.
<code>re_match_all()</code>	Find all regular expression matches in a string.
<code>re_replace()</code>	Replace matching substrings in a given string.
<code>re_test()</code>	Test if a regular expression has a match in a string.

---

**type str****Index Keywords** continuation cont

A unicode string of text.

Any other type (except `bytes`) can be `cast` to and from a string:

```
db> select <str>42;
{'42'}
db> select <bool>'true';
{true}
db> select "I EdgeDB";
{'I EdgeDB'}
```

Note that when a `str` is cast into a `json`, the result is a JSON string value. Same applies for casting back from `json` - only a JSON string value can be cast into a `str`:

```
db> select <json>'Hello, world';
{"Hello, world"}
```

There are two kinds of string literals in EdgeQL: regular and *raw*. Raw string literals do not evaluate \, so \n in a raw string is two characters \ and n.The regular string literal syntax is 'a string' or a "a string". Two *raw* string syntaxes are illustrated below:

```
db> select r'a raw \\ string';
{'a raw \\ string'}
db> select $$something$$;
{'something'}
db> select $marker$something $$
... nested \!$$$marker$;
{'something $$
nested \!$$'}
```

Regular strings use \ to indicate line continuation. When a line continuation symbol is encountered, the symbol itself as well as all the whitespace characters up to the next non-whitespace character are omitted from the string:

```
db> select 'Hello, \
...         world';
{"Hello, world"}
```

---

**Note:** This type is subject to the Postgres maximum field size of 1GB.

---

### operator str [ int64 ] -> str

String indexing.

Indexing starts at 0. Negative indexes are also valid and count from the *end* of the string.

```
db> select 'some text'[1];
{'o'}
db> select 'some text'[-1];
{'t'}
```

It is an error to attempt to extract a character at an index outside the bounds of the string:

```
db> select 'some text'[10];
InvalidValueError: string index 10 is out of bounds
```

---

### operator str [ int64 : int64 ] -> str

String slicing.

Indexing starts at 0. Negative indexes are also valid and count from the *end* of the string.

```
db> select 'some text'[1:3];
{'om'}
db> select 'some text'[-4:];
{'text'}
db> select 'some text'[:-5];
{'some'}
db> select 'some text'[5:-2];
{'te'}
```

It is perfectly acceptable to use indexes outside the bounds of a string in a *slice*:

```
db> select 'some text'[-4:100];
{'text'}
db> select 'some text'[-100:-5];
{'some'}
```

**operator str ++ str -> str**

String concatenation.

```
db> select 'some' ++ ' text';
{'some text'}
```

**operator str like str -> bool****operator str not like str -> bool**

Case-sensitive simple string matching.

Returns `true` if the *value* V matches the *pattern* P and `false` otherwise. The operator `not like` is the negation of `like`.

The pattern matching rules are as follows:

pattern	interpretation
<code>%</code>	matches zero or more characters
<code>_</code>	matches exactly one character
<code>\%</code>	matches a literal “%”
<code>\_</code>	matches a literal “_”
any other character	matches itself

In particular, this means that if there are no special symbols in the *pattern*, the operators `like` and `not like` work identical to `=` and `!=`, respectively.

```
db> select 'abc' like 'abc';
{true}
db> select 'abc' like 'a%';
{true}
db> select 'abc' like '_b_';
{true}
db> select 'abc' like 'c';
{false}
db> select 'a%%c' not like r'a\%c';
{true}
```

**operator str ilike str -> bool****operator str not ilike str -> bool**

Case-insensitive simple string matching.

The operators `ilike` and `not ilike` work the same way as `like` and `not like`, except that the *pattern* is matched in a case-insensitive manner.

```
db> select 'Abc' ilike 'a%';
{true}
```

---

**function std::str\_lower** → str

Return a lowercase copy of the input *string*.

```
db> select str_lower('Some Fancy Title');
{ 'some fancy title'}
```

---

**function std::str\_upper** → str

Return an uppercase copy of the input *string*.

```
db> select str_upper('Some Fancy Title');
{ 'SOME FANCY TITLE'}
```

---

**function std::str\_title** → str

Return a titlecase copy of the input *string*.

Every word in the *string* will have the first letter capitalized and the rest converted to lowercase.

```
db> select str_title('sOmE fAnCy TiTLE');
{ 'Some Fancy Title'}
```

---

**function std::str\_pad\_start** → str

Return the input *string* padded at the start to the length *n*.

If the *string* is longer than *n*, then it is truncated to the first *n* characters. Otherwise, the *string* is padded on the left up to the total length *n* using *fill* characters (space by default).

```
db> select str_pad_start('short', 10);
{ '      short'}
db> select str_pad_start('much too long', 10);
{ 'much too l'}
db> select str_pad_start('short', 10, '.:');
{ '....short'}
```

---

**function std::str\_pad\_end** → str

Return the input *string* padded at the end to the length *n*.

If the *string* is longer than *n*, then it is truncated to the first *n* characters. Otherwise, the *string* is padded on the right up to the total length *n* using *fill* characters (space by default).

```
db> select str_pad_end('short', 10);
{ 'short      '}
db> select str_pad_end('much too long', 10);
{ 'much too l'}
db> select str_pad_end('short', 10, '.:');
{ 'short.....'}
```

---

**function std::str\_trim\_start → str**

Return the input string with all *trim* characters removed from its start.

If the *trim* specifies more than one character they will be removed from the beginning of the *string* regardless of the order in which they appear.

```
db> select str_trim_start('      data');
{'data'}
db> select str_trim_start('.....data', '.:');
{'data'}
db> select str_trim_start(':::::data', '.:');
{'data'}
db> select str_trim_start('....:data', '.:');
{'data'}
db> select str_trim_start('.::::data', '.:');
{'data'}
```

**function std::str\_trim\_end → str**

Return the input string with all *trim* characters removed from its end.

If the *trim* specifies more than one character they will be removed from the end of the *string* regardless of the order in which they appear.

```
db> select str_trim_end('data      ');
{'data'}
db> select str_trim_end('data.....', '.:');
{'data'}
db> select str_trim_end('data:::::', '.:');
{'data'}
db> select str_trim_end('data....:', '.:');
{'data'}
db> select str_trim_end('data....:', '.:');
{'data'}
```

**function std::str\_trim → str**

Return the input string with *trim* characters removed from both ends.

If the *trim* specifies more than one character they will be removed from both ends of the *string* regardless of the order in which they appear. This is the same as applying `str_ltrim()` and `str_rtrim()`.

```
db> select str_trim('  data      ');
{'data'}
db> select str_trim('::data.....', '.:');
{'data'}
db> select str_trim('..data:::::', '.:');
{'data'}
db> select str_trim('..:data....:', '.:');
{'data'}
db> select str_trim('...:data...', '.:');
{'data'}
```

**function std::str\_repeat** → str

Repeat the input *string* *n* times.

If *n* is zero or negative an empty string is returned.

```
db> select str_repeat('. ', 3);
{'. . .'}
db> select str_repeat('foo', -1);
{''}
```

**function std::str\_replace** → str

Replace all occurrences of *old* substring with the *new* one.

Given a string *s* find all non-overlapping occurrences of the substring *old* and replace them with the substring *new*.

```
db> select str_replace('hello world', 'h', 'H');
{'Hello world'}
db> select str_replace('hello world', 'l', '[L]');
{'he[L] [L]o wor[L]d'}
db> select str_replace('hello world', 'o', '');
{'hell wrld'}
```

**function std::str\_reverse** → str

Reverse the order of the characters in the string.

```
db> select str_reverse('Hello world');
{'dlrow olleH'}
db> select str_reverse('Hello world ');
{' dlrow olleH'}
```

**function std::str\_split** → array<str>

**Index Keywords** split str\_split explode

Split string into array elements using the supplied delimiter.

```
db> select str_split('1, 2, 3', ', ');
{['1', '2', '3']}
```

```
db> select str_split('123', '');
{['1', '2', '3']}
```

**function std::re\_match** → array<str>

**Index Keywords** regex regexp regular

Find the first regular expression match in a string.

Given an input *string* and a regular expression *pattern* find the first match for the regular expression within the *string*. Return the match, each match represented by an *array<str>* of matched groups.

```
db> select re_match(r'\w{4}ql', 'I edgeql');
{['edgeql']}
```

**function std::re\_match\_all** → set of array<str>

**Index Keywords** regex regexp regular

Find all regular expression matches in a string.

Given an input *string* and a regular expression *pattern* repeatedly match the regular expression within the *string*. Return the set of all matches, each match represented by an *array<str>* of matched groups.

```
db> select re_match_all(r'a\w+', 'an abstract concept');
{['an'], ['abstract']}
```

**function std::re\_replace** → str

**Index Keywords** regex regexp regular replace

Replace matching substrings in a given string.

Given an input *string* and a regular expression *pattern* replace matching substrings with the replacement string *sub*. Optional *flag* arguments can be used to specify additional regular expression flags. Return the string resulting from substring replacement.

```
db> select re_replace(r'l', r'L', 'Hello World',
...                      flags := 'g');
{'HeLLo WorLd'}
```

**function std::re\_test** → bool

**Index Keywords** regex regexp regular match

Test if a regular expression has a match in a string.

Given an input *string* and a regular expression *pattern* test whether there is a match for the regular expression within the *string*. Return `true` if there is a match, `false` otherwise.

```
db> select re_test(r'a', 'abc');
{true}
```

**function std::to\_str** → str  
**function std::to\_str** → str

**Index Keywords** stringify dumps join array\_to\_string

Return string representation of the input value.

This is a very versatile polymorphic function that is defined for many different input types. In general, there are corresponding converter functions from `str` back to the specific types, which share the meaning of the format argument `fmt`.

When converting `datetime`, `cal::local_datetime`, `cal::local_date`, `cal::local_time`, `duration` this function is the inverse of `to_datetime()`, `cal::to_local_datetime()`, `cal::to_local_date()`, `cal::to_local_time()`, `to_duration()`, correspondingly.

For valid date and time formatting patterns see [here](#).

```
db> select to_str(<datetime>'2018-05-07 15:01:22.306916-05',
...                      'FMDdth of FMMonth, YYYY');
{ '7th of May, 2018' }
db> select to_str(<cal::local_date>'2018-05-07', 'CCth "century"');
{ '21st century' }
```

When converting one of the numeric types, this function is the reverse of: `to_bigint()`, `to_decimal()`, `to_int16()`, `to_int32()`, `to_int64()`, `to_float32()`, `to_float64()`.

For valid number formatting patterns see [here](#).

See also `to_json()`.

```
db> select to_str(123, '999999');
{ ' 123' }
db> select to_str(123, '099999');
{ ' 000123' }
db> select to_str(123.45, 'S999.999');
{ '+123.450' }
db> select to_str(123.45e-20, '9.99EEEE');
{ ' 1.23e-18' }
db> select to_str(-123.45n, 'S999.99');
{ '-123.45' }
```

When converting `json`, this function can take 'pretty' as the optional `fmt` argument to produce a pretty-formatted JSON string.

See also `to_json()`.

```
db> select to_str(<json>2);
{ '2' }

db> select to_str(<json>['hello', 'world']);
{ '['"hello", "world"]' }

db> select to_str(<json>(a := 2, b := 'hello'), 'pretty');
{ '{\n    "a": 2,\n    "b": "hello"\n}' }
```

When converting `arrays`, a `delimiter` argument is required:

```
db> select to_str(['one', 'two', 'three'], ', ', ',');
{ 'one, two, three' }
```

**Warning:** There's a deprecated version of `std::to_str` which operates on arrays, however `array_join()` should be used instead.

## 5.5.1 Regular Expressions

EdgeDB supports Regular expressions (REs), as defined in POSIX 1003.2. They come in two forms: BRE (basic RE) and ERE (extended RE). In addition, EdgeDB supports certain common extensions to the POSIX standard commonly known as ARE (advanced RE). More details about BRE, ERE, and ARE support can be found in [PostgreSQL documentation](#).

For convenience, here's a table outlining the different options accepted as the `flags` argument to various regular expression functions, or as [embedded options](#) in the pattern itself, e.g. `'(?i)fooBAR'`:

### 5.5.1.1 Option Flags

Option	Description
b	rest of RE is a BRE
c	case-sensitive matching (overrides operator type)
e	rest of RE is an ERE
i	case-insensitive matching (overrides operator type)
m	historical synonym for n
n	newline-sensitive matching
p	partial newline-sensitive matching
q	rest of RE is a literal (“quoted”) string, all ordinary characters
s	non-newline-sensitive matching (default)
t	tight syntax (default)
w	inverse partial newline-sensitive (“weird”) matching
x	expanded syntax ignoring white-space characters

## 5.5.2 Formatting

Some of the type converter functions take an extra argument specifying the formatting (either for converting to a `str` or parsing from one). The different formatting options are collected in this section.

### 5.5.2.1 Date and time formatting options

Pattern	Description
HH	hour of day (01-12)
HH12	hour of day (01-12)
HH24	hour of day (00-23)
MI	minute (00-59)
SS	second (00-59)

continues on next page

Table 1 – continued from previous page

Pattern	Description
MS	millisecond (000-999)
US	microsecond (000000-999999)
SSSS	seconds past midnight (0-86399)
AM, am, PM or pm	meridiem indicator (without periods)
A.M., a.m., P.M. or p.m.	meridiem indicator (with periods)
Y,YYY	year (4 or more digits) with comma
YYYY	year (4 or more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
IYYY	ISO 8601 week-numbering year (4 or more digits)
IYY	last 3 digits of ISO 8601 week- numbering year
IY	last 2 digits of ISO 8601 week- numbering year
I	last digit of ISO 8601 week-numbering year
BC, bc, AD or ad	era indicator (without periods)
B.C., b.c., A.D. or a.d.	era indicator (with periods)
MONTH	full upper case month name (blank- padded to 9 chars)
Month	full capitalized month name (blank- padded to 9 chars)
month	full lower case month name (blank- padded to 9 chars)
MON	abbreviated upper case month name (3 chars in English, localized lengths vary)
Mon	abbreviated capitalized month name (3 chars in English, localized lengths vary)
mon	abbreviated lower case month name (3 chars in English, localized lengths vary)
MM	month number (01-12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full capitalized day name (blank- padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars in English, localized lengths vary)
Dy	abbreviated capitalized day name (3 chars in English, localized lengths vary)
dy	abbreviated lower case day name (3 chars in English, localized lengths vary)
DDD	day of year (001-366)
IDDD	day of ISO 8601 week-numbering year (001-371; day 1 of the year is Monday of the first ISO week)
DD	day of month (01-31)
D	day of the week, Sunday (1) to Saturday (7)
ID	ISO 8601 day of the week, Monday (1) to Sunday (7)
W	week of month (1-5) (the first week starts on the first day of the month)
WW	week number of year (1-53) (the first week starts on the first day of the year)
IW	week number of ISO 8601 week-numbering year (01-53; the first Thursday of the year is in week 1)
CC	century (2 digits) (the twenty-first century starts on 2001-01-01)
J	Julian Day (integer days since November 24, 4714 BC at midnight UTC)
Q	quarter
RM	month in upper case Roman numerals (I-XII; I=January)
rm	month in lower case Roman numerals (i-xii; i=January)
TZ	upper case time-zone abbreviation (only supported in to_char)
tz	lower case time-zone abbreviation (only supported in to_char)
TZH	time-zone hours
TZM	time-zone minutes
OF	time-zone offset from UTC (only supported in to_char)

Some additional formatting modifiers:

Modifier	Description	Example
FM prefix	fill mode (suppress leading zeroes and padding blanks)	FMMonth
TH suffix	upper case ordinal number suffix	DDTH, e.g., 12TH
th suffix	lower case ordinal number suffix	DDth, e.g., 12th
FX prefix	fixed format global option (see usage notes)	FX Month DD Day

Normally when parsing a string input whitespace is ignored, unless the *FX* prefix modifier is used. For example:

```
db> select cal::to_local_date(
...     '2000      JUN', 'YYYY MON');
{<cal::local_date>'2000-06-01'}
db> select cal::to_local_date(
...     '2000      JUN', 'FXYYYY MON');
InternalServerError: invalid value "      " for "MON"
```

### 5.5.2.2 Number formatting options

Pattern	Description
9	digit position (can be dropped if insignificant)
0	digit position (will not be dropped, even if insignificant)
.	(period) decimal point
,	(comma) group (thousands) separator
PR	negative value in angle brackets
S	sign anchored to number (uses locale)
L	currency symbol (uses locale)
D	decimal point (uses locale)
G	group separator (uses locale)
MI	minus sign in specified position (if number < 0)
PL	plus sign in specified position (if number > 0)
SG	plus/minus sign in specified position
RN	Roman numeral (input between 1 and 3999)
TH or th	ordinal number suffix
V	shift specified number of digits (see notes)
EEEE	exponent for scientific notation

Some additional formatting modifiers:

Modifier	Description	Example
FM prefix	fill mode (suppress leading zeroes and padding blanks)	FM99.99
TH suffix	upper case ordinal number suffix	999TH
th suffix	lower case ordinal number suffix	999th

## 5.6 Booleans

**edb-alt-title** Boolean Functions and Operators

<code>bool</code>	Boolean type
<code>bool or bool</code>	Evaluates true if either boolean is true.
<code>bool and bool</code>	Evaluates true if both booleans are true.
<code>not bool</code>	Logically negates a given boolean value.
<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>all()</code>	Returns true if none of the values in the given set are false.
<code>any()</code>	Returns true if any of the values in the given set is true.
<code>assert()</code>	Checks that the input bool is true.

---

### **type** `bool`

A boolean type of either `true` or `false`.

EdgeQL has case-insensitive keywords and that includes the boolean literals:

```
db> select (True, true, TRUE);  
{true, true, true}  
db> select (False, false, FALSE);  
{false, false, false}
```

These basic operators will always result in a boolean value:

- `=`
- `!=`
- `?=`
- `?!=`
- `in`
- `not in`
- `<`
- `>`
- `<=`
- `>=`
- `like`
- `ilike`

Some examples:

```
db> select true and 2 < 3;  
{true}  
db> select '!' IN {'hello', 'world'};  
{false}
```

It's possible to get a boolean by casting a `str` or `json` value into it:

```
db> select <bool>('true');
{true}
db> select <bool>to_json('false');
{false}
```

*Filter clauses* must always evaluate to a boolean:

```
select User
filter .name ilike 'alice';
```

#### operator bool or bool -> bool

Evaluates true if either boolean is true.

```
db> select false or true;
{true}
```

#### operator bool and bool -> bool

Evaluates true if both booleans are true.

```
db> select false and true;
{false}
```

#### operator not bool -> bool

Logically negates a given boolean value.

```
db> select not false;
{true}
```

The and and or operators are commutative.

The truth tables are as follows:

a	b	a and b	a or b	not a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

The operators and/or and the functions `all()`/`any()` differ in the way they handle an empty set (`{}`). Both and and or operators apply to the cross-product of their operands. If either operand is an empty set, the result will also be an empty set. For example:

```
db> select {true, false} and <bool>{};
{}
db> select true and <bool>{};
{}
```

Operating on an empty set with `all()`/`any()` does *not* return an empty set:

```
db> select all(<bool>{});  
{true}  
db> select any(<bool>{});  
{false}
```

`all()` returns `true` because the empty set contains no false values.

`any()` returns `false` because the empty set contains no true values.

The `all()` and `any()` functions are generalized to apply to sets of values, including `{}`. Thus they have the following truth table:

a	b	all({a, b})	any({a, b})
true	true	true	true
true	false	false	true
{}	true	true	true
{}	false	false	false
false	true	false	true
false	false	false	false
true	{}	true	true
false	{}	false	false
{}	{}	true	false

Since `all()` and `any()` apply to sets as a whole, missing values (represented by `{}`) are just that - missing. They don't affect the overall result.

To understand the last line in the above truth table it's useful to remember that `all({a, b}) = all(a) and all(b)` and `any({a, b}) = any(a) or any(b)`.

For more customized handling of `{}`, use the `??` operator.

---

#### **function std::assert → bool**

Checks that the input bool is `true`.

If the input bool is `false`, `assert` raises a `QueryAssertionError`. Otherwise, this function returns `true`.

```
db> select assert(true);  
{true}  
  
db> select assert(false);  
edgedb error: QueryAssertionError: assertion failed  
  
db> select assert(false, message := 'value is not true');  
edgedb error: QueryAssertionError: value is not true
```

`assert` can be used in triggers to create more powerful constraints. In this schema, the `Person` type has both `friends` and `enemies` links. You may not want a `Person` to be both a friend and an enemy of the same `Person`. `assert` can be used inside a trigger to easily prohibit this.

```
type Person {  
    required name: str;  
    multi friends: User;
```

(continues on next page)

(continued from previous page)

```

multi enemies: User;

trigger prohibit_frenemies after insert, update for each do (
    assert(
        not exists (__new__.friends intersect __new__.enemies),
        message := "Invalid frenemies",
    )
)
}

```

With this trigger in place, it is impossible to link the same Person as both a friend and an enemy of any other person.

```

db> insert Person {name := 'Quincey Morris'};
{default::Person {id: e4a55480-d2de-11ed-93bd-9f4224fc73af}}
db> insert Person {name := 'Dracula'};
{default::Person {id: e7f2cff0-d2de-11ed-93bd-279780478afb}}
db> update User
... filter .name = 'Quincey Morris'
... set {
...     enemies := (select Person filter .name = 'Dracula')
... };
{default::Person {id: e4a55480-d2de-11ed-93bd-9f4224fc73af}}
db> update User
... filter .name = 'Quincey Morris'
... set {
...     friends := (select Person filter .name = 'Dracula')
... };
edgedb error: EdgeDBError: Invalid frenemies

```

In the following examples, the `size` properties of the `File` objects are `1024`, `1024`, and `131,072`.

```

db> for obj in (select File)
... union (assert(obj.size <= 128*1024, message := 'file too big'));
{true, true, true}

db> for obj in (select File)
... union (assert(obj.size <= 64*1024, message := 'file too big'));
edgedb error: QueryAssertionError: file too big

```

You may call `assert` in the `order by` clause of your `select` statement. This will ensure it is called only on objects that pass your filter.

```

db> select File { name, size }
... order by assert(.size <= 128*1024, message := "file too big");
{
    default::File {name: 'File 2', size: 1024},
    default::File {name: 'Asdf 3', size: 1024},
    default::File {name: 'File 1', size: 131072},
}

db> select File { name, size }
... order by assert(.size <= 64*1024, message := "file too big");

```

(continues on next page)

(continued from previous page)

```
edgedb error: QueryAssertionError: file too big

db> select File { name, size }
... filter .size <= 64*1024
... order by assert(.size <= 64*1024, message := "file too big");
{
    default::File {name: 'File 2', size: 1024},
    default::File {name: 'Asdf 3', size: 1024}
}
```

## 5.7 Numbers

**edb-alt-title** Numerical Types, Functions, and Operators

<i>int16</i>	16-bit integer
<i>int32</i>	32-bit integer
<i>int64</i>	64-bit integer
<i>float32</i>	32-bit floating point number
<i>float64</i>	64-bit floating point number
<i>bigint</i>	Arbitrary precision integer.
<i>decimal</i>	Arbitrary precision number.
<i>anyreal + anyreal</i>	Arithmetic addition.
<i>anyreal - anyreal</i>	Arithmetic subtraction.
<i>-anyreal</i>	Arithmetic negation.
<i>anyreal * anyreal</i>	Arithmetic multiplication.
<i>anyreal / anyreal</i>	Arithmetic division.
<i>anyreal // anyreal</i>	Floor division.
<i>anyreal % anyreal</i>	Remainder from division (modulo).
<i>anyreal ^ anyreal</i>	Power operation.
<i>= != ?= ?!= &lt; &gt; &lt;= &gt;=</i>	Comparison operators
<i>sum()</i>	Returns the sum of the set of numbers.
<i>min()</i>	Returns the smallest value in the given set.
<i>max()</i>	Returns the largest value in the given set.
<i>round()</i>	Rounds a given number to the nearest value.
<i>random()</i>	Returns a pseudo-random number in the range of 0.0 <= x < 1.0.

### 5.7.1 Mathematical functions

<code>math::abs()</code>	Returns the absolute value of the input.
<code>math::ceil()</code>	Rounds up a given value to the nearest integer.
<code>math::floor()</code>	Rounds down a given value to the nearest integer.
<code>math::ln()</code>	Returns the natural logarithm of a given value.
<code>math::lg()</code>	Returns the base 10 logarithm of a given value.
<code>math::log()</code>	Returns the logarithm of a given value in the specified base.
<code>math::mean()</code>	Returns the arithmetic mean of the input set.
<code>math::stddev()</code>	Returns the sample standard deviation of the input set.
<code>math::stddev_pop()</code>	Returns the population standard deviation of the input set.
<code>math::var()</code>	Returns the sample variance of the input set.
<code>math::var_pop()</code>	Returns the population variance of the input set.

### 5.7.2 Bitwise functions

<code>bit_and()</code>	Bitwise AND operator for 2 intergers.
<code>bit_or()</code>	Bitwise OR operator for 2 intergers.
<code>bit_xor()</code>	Bitwise exclusive OR operator for 2 intergers.
<code>bit_not()</code>	Bitwise negation operator for 2 intergers.
<code>bit_lshift()</code>	Bitwise left-shift operator for intergers.
<code>bit_rshift()</code>	Bitwise arithmetic right-shift operator for intergers.

### 5.7.3 String parsing

<code>to_bigint()</code>	Returns a bigint value parsed from the given string.
<code>to_decimal()</code>	Returns a decimal value parsed from the given string.
<code>to_int16()</code>	Returns an int16 value parsed from the given string.
<code>to_int32()</code>	Returns an int32 value parsed from the given string.
<code>to_int64()</code>	Returns an int64 value parsed from the given string.
<code>to_float32()</code>	Returns a float32 value parsed from the given string.
<code>to_float64()</code>	Returns a float64 value parsed from the given string.

It's possible to explicitly `cast` between all numeric types. All numeric types can also be cast to and from `str` and `json`.

#### type int16

**Index Keywords** int integer

A 16-bit signed integer.

`int16` is capable of representing values from -32768 to +32767 (inclusive).

#### type int32

**Index Keywords** int integer

A 32-bit signed integer.

`int32` is capable of representing values from -2147483648 to +2147483647 (inclusive).

---

**type `int64`**

**Index Keywords** `int` `integer`

A 64-bit signed integer.

`int64` is capable of representing values from -9223372036854775808 to +9223372036854775807 (inclusive).

---

**type `float32`**

**Index Keywords** `float`

A variable precision, inexact number.

The minimal guaranteed precision is at least 6 decimal digits. The approximate range of a `float32` spans from -3.4e+38 to +3.4e+38.

---

**type `float64`**

**Index Keywords** `float` `double`

A variable precision, inexact number.

The minimal guaranteed precision is at least 15 decimal digits. The approximate range of a `float64` spans from -1.7e+308 to +1.7e+308.

---

**type `bigint`**

**Index Keywords** `numeric` `bigint`

An arbitrary precision integer.

Our philosophy is that use of `bigint` should always be an explicit opt-in and should never be implicit. Once used, these values should not be accidentally cast to a different numerical type that could lead to a loss of precision.

In keeping with this philosophy, *our mathematical functions* are designed to maintain separation between big integer values and the rest of our numeric types.

All of the following types can be explicitly cast into a `bigint` type:

- `str`
- `json`
- `int16`
- `int32`
- `int64`
- `float32`
- `float64`
- `decimal`

A bigint literal is an integer literal, followed by ‘n’:

```
db> select 42n is bigint;
{true}
```

To represent really big integers, it is possible to use the exponent notation (e.g. `1e20n` instead of `1000000000000000000000000n`) as long as the exponent is positive and there is no dot anywhere:

```
db> select 1e+100n is bigint;
{true}
```

When a float literal is followed by n it will produce a `decimal` value instead:

```
db> select 1.23n is decimal;
{true}

db> select 1.0e+100n is decimal;
{true}
```

---

**Note:** Use caution when casting `bigint` values into `json`. The JSON specification does not have a limit on significant digits, so a bigint number can be losslessly represented in JSON. However, JSON decoders in many languages will read all such numbers as some kind of 32-bit or 64-bit number type, which may result in errors or precision loss. If such loss is unacceptable, then consider casting the value into `str` and decoding it on the client side into a more appropriate type.

---

## type decimal

**Index Keywords** numeric float

Any number of arbitrary precision.

Our philosophy is that use of `decimal` should always be an explicit opt-in and should never be implicit. Once used, these values should not be accidentally cast to a different numerical type that could lead to a loss of precision.

In keeping with this philosophy, *our mathematical functions* are designed to maintain separation between `decimal` values and the rest of our numeric types.

All of the following types can be explicitly cast into `decimal`:

- `str`
- `json`
- `int16`
- `int32`
- `int64`
- `float32`
- `float64`
- `bigint`

A decimal literal is a float literal, followed by n:

The EdgeDB philosophy is that using a decimal type should be an explicit opt-in, but once used, the values should not be accidentally cast into a numeric type with less precision.

In accordance with this [the mathematical functions](#) are designed to keep the separation between decimal values and the rest of the numeric types.

All of the following types can be explicitly cast into decimal: `str`, `json`, `int16`, `int32`, `int64`, `float32`, `float64`, and `bigint`.

A decimal literal is a float literal followed by ‘n’:

```
db> select 1.23n is decimal;
{true}

db> select 1.0e+100n is decimal;
{true}
```

Note that an integer literal (without a dot or exponent) followed by n produces a `bigint` value. A literal without a dot and with a positive exponent makes a `bigint`, too:

```
db> select 42n is bigint;
{true}

db> select 12e+34n is bigint;
{true}
```

---

**Note:** Use caution when casting decimal values into `json`. The JSON specification does not have a limit on significant digits, so a decimal number can be losslessly represented in JSON. However, JSON decoders in many languages will read all such numbers as some kind of floating point values, which may result in precision loss. If such loss is unacceptable, then consider casting the value into a `str` and decoding it on the client side into a more appropriate type.

---

### operator anyreal + anyreal -> anyreal

**Index Keywords** plus add

Arithmetic addition.

```
db> select 2 + 2;
{4}
```

---

### operator anyreal - anyreal -> anyreal

**Index Keywords** minus subtract

Arithmetic subtraction.

```
db> select 3 - 2;
{1}
```

---

### operator - anyreal -> anyreal

---

**Index Keywords** unary minus subtract

Arithmetic negation.

```
db> select -5;
{-5}
```

---

**operator anyreal \* anyreal -> anyreal**

**Index Keywords** multiply multiplication

Arithmetic multiplication.

```
db> select 2 * 10;
{20}
```

---

**operator anyreal / anyreal -> anyreal**

**Index Keywords** divide division

Arithmetic division.

```
db> select 10 / 4;
{2.5}
```

Division by zero will result in an error:

```
db> select 10 / 0;
DivisionByZeroError: division by zero
```

---

**operator anyreal // anyreal -> anyreal**

**Index Keywords** floor divide division

Floor division.

In floor-based division, the result of a standard division operation is rounded down to its nearest integer. It is the equivalent to using regular division and then applying `math::floor()` to the result.

```
db> select 10 // 4;
{2}
db> select math::floor(10 / 4);
{2}
db> select -10 // 4;
{-3}
```

It also works on `float`, `bigint`, and `decimal` types. The type of the result corresponds to the type of the operands:

```
db> select 3.7 // 1.1;
{3.0}
db> select 3.7n // 1.1n;
{3.0n}
db> select 37 // 11;
{3}
```

Regular division, floor division, and `%` operations are related in the following way:  $A // B = (A - (A \% B)) / B$ .

---

### operator anyreal % anyreal -> anyreal

**Index Keywords** modulo mod division

Remainder from division (modulo).

This is commonly referred to as a “modulo” operation.

This is the remainder from floor division. Just as is the case with `//` the result type of the remainder operator corresponds to the operand type:

```
db> select 10 % 4;
{2}
db> select 10n % 4;
{2n}
db> select -10 % 4;
{2}
db> # floating arithmetic is inexact, so
... # we get 0.399999999999999 instead of 0.4
... select 3.7 % 1.1;
{0.399999999999999}
db> select 3.7n % 1.1n;
{0.4n}
db> select 37 % 11;
{4}
```

Regular division, `//` and `%` operations are related in the following way:  $A // B = (A - (A \% B)) / B$ .

Modulo division by zero will result in an error:

```
db> select 10 % 0;
DivisionByZeroError: division by zero
```

---

### operator anyreal ^ anyreal -> anyreal

**Index Keywords** power pow

Power operation.

```
db> select 2 ^ 4;
{16}
```

---

```
function std::round → float64
function std::round → float64
function std::round → bigint
function std::round → decimal
function std::round → decimal
```

Rounds a given number to the nearest value.

The function will round a `.5` value differently depending on the type of the parameter passed.

The `float64` tie is rounded to the nearest even number:

```
db> select round(1.2);
{1}

db> select round(1.5);
{2}

db> select round(2.5);
{2}
```

But the *decimal* tie is rounded away from zero:

```
db> select round(1.2n);
{1n}

db> select round(1.5n);
{2n}

db> select round(2.5n);
{3n}
```

Additionally, when rounding a *decimal* value, you may pass the optional argument *d* to specify the precision of the rounded result:

```
db> select round(163.278n, 2);
{163.28n}

db> select round(163.278n, 1);
{163.3n}

db> select round(163.278n, 0);
{163n}

db> select round(163.278n, -1);
{160n}

db> select round(163.278n, -2);
{200n}
```

### **function std::random** → float64

Returns a pseudo-random number in the range  $0.0 \leq x < 1.0$ .

```
db> select random();
{0.62649393780157}
```

### **function std::bit\_and** → int16

### **function std::bit\_and** → int32

### **function std::bit\_and** → int64

Bitwise AND operator for 2 intergers.

```
db> select bit_and(17, 3);
{1}
```

---

```
function std::bit_or → int16
function std::bit_or → int32
function std::bit_or → int64
Bitwise OR operator for 2 intergers.
```

```
db> select bit_or(17, 3);
{19}
```

---

```
function std::bit_xor → int16
function std::bit_xor → int32
function std::bit_xor → int64
Bitwise exclusive OR operator for 2 intergers.
```

```
db> select bit_xor(17, 3);
{18}
```

---

```
function std::bit_not → int16
function std::bit_not → int32
function std::bit_not → int64
Bitwise negation operator for 2 intergers.
```

Bitwise negation for integers ends up similar to mathematical negation because typically the signed integers use “two’s complement” representation. In this representation mathematical negation is achieved by applying bitwise negation and adding 1.

```
db> select bit_not(17);
{-18}
db> select -17 = bit_not(17) + 1;
{true}
```

---

```
function std::bit_lshift → int16
function std::bit_lshift → int32
function std::bit_lshift → int64
Bitwise left-shift operator for intergers.
```

The integer *val* is shifted by *n* bits to the left. The rightmost added bits are all **0**. Shifting an integer by a number of bits greater than the bit size of the integer results in **0**.

```
db> select bit_lshift(123, 2);
{492}
db> select bit_lshift(123, 65);
{0}
```

Left-shifting an integer can change the sign bit:

```
db> select bit_lshift(123, 60);
{-5764607523034234880}
```

In general, left-shifting an integer in small increments produces the same result as shifting it in one step:

```
db> select bit_lshift(bit_lshift(123, 1), 3);
{1968}
db> select bit_lshift(123, 4);
{1968}
```

It is an error to attempt to shift by a negative number of bits:

```
db> select bit_lshift(123, -2);
edgedb error: InvalidValueError: bit_lshift(): cannot shift by
negative amount
```

```
function std::bit_rshift → int16
function std::bit_rshift → int32
function std::bit_rshift → int64
```

Bitwise arithmetic right-shift operator for integers.

The integer *val* is shifted by *n* bits to the right. In the arithmetic right-shift, the sign is preserved. This means that the leftmost added bits are 1 or 0 depending on the sign bit. Shifting an integer by a number of bits greater than the bit size of the integer results in 0 for positive numbers or -1 for negative numbers.

```
db> select bit_rshift(123, 2);
{30}
db> select bit_rshift(123, 65);
{0}
db> select bit_rshift(-123, 2);
{-31}
db> select bit_rshift(-123, 65);
{-1}
```

In general, right-shifting an integer in small increments produces the same result as shifting it in one step:

```
db> select bit_rshift(bit_rshift(123, 1), 3);
{7}
db> select bit_rshift(123, 4);
{7}
db> select bit_rshift(bit_rshift(-123, 1), 3);
{-8}
db> select bit_rshift(-123, 4);
{-8}
```

It is an error to attempt to shift by a negative number of bits:

```
db> select bit_rshift(123, -2);
edgedb error: InvalidValueError: bit_rshift(): cannot shift by
negative amount
```

```
function std::to_bigint → bigint
```

**Index Keywords** parse bigint

Returns a *bigint* value parsed from the given string.

The function will use an optional format string passed as *fmt*. See the [number formatting options](#) for help writing a format string.

```
db> select to_bigint('-000,012,345', 'S099,999,999,999');
{-12345n}
db> select to_bigint('31st', '999th');
{31n}
```

---

**function std::to\_decimal** → decimal

**Index Keywords** parse decimal

Returns a `decimal` value parsed from the given string.

The function will use an optional format string passed as `fmt`. See the [number formatting options](#) for help writing a format string.

```
db> select to_decimal('-000,012,345', 'S099,999,999,999');
{-12345.0n}
db> select to_decimal('-012.345');
{-12.345n}
db> select to_decimal('31st', '999th');
{31.0n}
```

---

**function std::to\_int16** → int16

**Index Keywords** parse int16

Returns an `int16` value parsed from the given string.

The function will use an optional format string passed as `fmt`. See the [number formatting options](#) for help writing a format string.

---

**function std::to\_int32** → int32

**Index Keywords** parse int32

Returns an `int32` value parsed from the given string.

The function will use an optional format string passed as `fmt`. See the [number formatting options](#) for help writing a format string.

---

**function std::to\_int64** → int64

**Index Keywords** parse int64

Returns an `int64` value parsed from the given string.

The function will use an optional format string passed as `fmt`. See the [number formatting options](#) for help writing a format string.

---

**function std::to\_float32** → float32

**Index Keywords** parse float32

Returns a `float32` value parsed from the given string.

The function will use an optional format string passed as `fmt`. See the [number formatting options](#) for help writing a format string.

```
function std::to_float64 → float64
```

**Index Keywords** `parse` `float64`

Returns a `float64` value parsed from the given string.

The function will use an optional format string passed as `fmt`. See the [number formatting options](#) for help writing a format string.

## 5.8 JSON

### edb-alt-title JSON Functions and Operators

<code>json</code>	JSON scalar type
<code>json[i]</code>	Accesses the element of the JSON string or array at a given index.
<code>json[from:to]</code>	Produces a JSON value comprising a portion of the existing JSON value.
<code>json ++ json</code>	Concatenates two JSON arrays, objects, or strings into one.
<code>json[name]</code>	Accesses an element of a JSON object given its key.
<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>to_json()</code>	Returns a JSON value parsed from the given string.
<code>to_str()</code>	Render JSON value to a string.
<code>json_get()</code>	Returns a value from a JSON object or array given its path.
<code>json_set()</code>	Returns an updated JSON target with a new value.
<code>json_array_unpack()</code>	Returns the elements of a JSON array as a set of <code>json</code> .
<code>json_object_pack()</code>	Returns the given set of key/value tuples as a JSON object.
<code>json_object_unpack()</code>	Returns the data in a JSON object as a set of key/value tuples.
<code>json_typeof()</code>	Returns the type of the outermost JSON value as a string.

### 5.8.1 Constructing JSON Values

JSON in EdgeDB is a [scalar type](#). This type doesn't have its own literal, and instead can be obtained by either casting a value to the `json` type, or by using the `to_json()` function:

```
db> select to_json('{"hello": "world"}');
{Json("{"hello": "world"})}
db> select <json>'hello world';
{Json("\\"hello world\\")}
```

Any value in EdgeDB can be cast to a `json` type as well:

```
db> select <json>2019;
{Json("2019")}
db> select <json>cal::to_local_date(datetime_current(), 'UTC');
{Json("\\"2022-11-21\\")}
```

The `json_object_pack()` function provides one more way to construct JSON. It constructs a JSON object from an array of key/value tuples:

```
db> select json_object_pack({{"hello", <json>"world"}});
{Json("{\"hello\": \"world\"")}
```

Additionally, any *Object* in EdgeDB can be cast as a *json* type. This produces the same JSON value as the JSON-serialized result of that said object. Furthermore, this result will be the same as the output of a *select expression* in *JSON mode*, including the shape of that type:

```
db> select <json>(
...     select schema::Object {
...         name,
...         timestamp := cal::to_local_date(
...             datetime_current(), 'UTC')
...     }
...     filter .name = 'std::bool');
{Json("{\"name\": \"std::bool\", \"timestamp\": \"2022-11-21\"")}
```

JSON values can also be cast back into scalars. Casting JSON is symmetrical meaning that, if a scalar value can be cast into JSON, a compatible JSON value can be cast into a scalar of that type. Some scalar types will have specific conditions for casting:

- JSON strings can be cast to a *str* type. Casting *uuid* and *date/time* types to JSON results in a JSON string representing its original value. This means it is also possible to cast a JSON string back to those types. The value of the UUID or datetime string must be properly formatted to successfully cast from JSON, otherwise EdgeDB will raise an exception.
- JSON numbers can be cast to any *numeric type*.
- JSON booleans can be cast to a *bool* type.
- JSON null is unique because it can be cast to an empty set ({} ) of any type.
- JSON arrays can be cast to any valid array type, as long as the JSON array is homogeneous, does not contain null as an element of the array, and does not contain another array.

A named *tuple* is converted into a JSON object when cast as a *json* while a standard *tuple* is converted into a JSON array. Unlike other casts to JSON, tuple casts to JSON are *not* reversible (i.e., it is not possible to cast a JSON value directly into a *tuple*).

---

## type json

Arbitrary JSON data.

Any other type can be *cast* to and from JSON:

```
db> select <json>42;
{Json("42")}
db> select <bool>to_json('true');
{true}
```

A *json* value can also be cast as a *str* type, but only when recognized as a JSON string:

```
db> select <str>to_json('"something"');
{'something'}
```

Casting a JSON array of strings (["a", "b", "c"]) to a *str* will result in an error:

```
db> select <str>to_json('["a", "b", "c"]');
InvalidValueError: expected json string or null; got JSON array
```

Instead, use the `to_str()` function to dump a JSON value to a `str` value. Use the `to_json()` function to parse a JSON string to a `json` value:

```
db> select to_json('[1, "a"]');
{Json("[1, \"a\"]")}
db> select to_str(<json>[1, 2]);
'[1, 2]'
```

---

**Note:** This type is backed by the Postgres `jsonb` type which has a size limit of 256MiB minus one byte. The EdgeDB `json` type is also subject to this limitation.

---

### operator `json [ int64 ] -> json`

Accesses the element of the JSON string or array at a given index.

The contents of JSON *arrays* and *strings* can also be accessed via `[]`:

```
db> select <json>'hello'[1];
{Json("\\"e\\")}
db> select <json>'hello'[-1];
{Json("\\"o\\")}
db> select to_json('[1, "a", null')[1];
{Json("\\"a\\")}
db> select to_json('[1, "a", null')[-1];
{Json("null")}
```

This will raise an exception if the specified index is not valid for the base JSON value. To access an index that is potentially out of bounds, use `json_get()`.

---

### operator `json [ int64 : int64 ] -> json`

Produces a JSON value comprising a portion of the existing JSON value.

JSON *arrays* and *strings* can be sliced in the same way as regular arrays, producing a new JSON array or string:

```
db> select <json>'hello'[0:2];
{Json("\\"he\\")}
db> select <json>'hello'[2:];
{Json("\\"llo\\")}
db> select to_json('[1, 2, 3')[0:2];
{Json("[1, 2]")}
db> select to_json('[1, 2, 3')[2:];
{Json("[3]")}
db> select to_json('[1, 2, 3')[::1];
{Json("[1]")}
db> select to_json('[1, 2, 3')[::-2];
{Json("[1]")}
```

**operator json ++ json -> json**

Concatenates two JSON arrays, objects, or strings into one.

JSON arrays, objects and strings can be concatenated with JSON values of the same type into a new JSON value.

If you concatenate two JSON objects, you get a new object whose keys will be a union of the keys of the input objects. If a key is present in both objects, the value from the second object is taken.

```
db> select to_json('[1, 2]') ++ to_json('[3]');
{Json("[1, 2, 3]")}
db> select to_json('{"a": 1}') ++ to_json('{"b": 2}');
{Json("{\"a\": 1, \"b\": 2}")}
db> select to_json('{"a": 1, "b": 2}') ++ to_json('{"b": 3}');
{Json("{\"a\": 1, \"b\": 3"})}
db> select to_json('"123") ++ to_json('"456");
{Json("\\"123456\\")}
```

**operator json [ str ] -> json**

Accesses an element of a JSON object given its key.

The fields of JSON *objects* can also be accessed via []:

```
db> select to_json('{"a": 2, "b": 5}')['b'];
{Json("5")}
db> select j := <json>(schema::Type {
    ...     name,
    ...     timestamp := cal::to_local_date(datetime_current(), 'UTC')
    ... })
... filter j['name'] = <json>'std::bool';
{Json("{\"name\": \"std::bool\", \"timestamp\": \"2022-11-21\""})}
```

This will raise an exception if the specified field does not exist for the base JSON value. To access an index that is potentially out of bounds, use `json_get()`.

---

**function std::to\_json → json**

**Index Keywords** json parse loads

Returns a JSON value parsed from the given string.

```
db> select to_json('[1, "hello", null]');
{Json("[1, \"hello\", null])}
db> select to_json('{"hello": "world"}');
{Json("{\"hello\": \"world\"})}
```

**function std::json\_array\_unpack → set of json**

**Index Keywords** array unpack

Returns the elements of a JSON array as a set of `json`.

Calling this function on anything other than a JSON array will result in a runtime error.

This function should be used only if the ordering of elements is not important, or when the ordering of the set is preserved (such as an immediate input to an aggregate function).

```
db> select json_array_unpack(to_json('[1, "a"]'));
{Json("1"), Json("\\"a\\")}
```

## function std::json\_get → optional json

**Index Keywords** safe navigation

Returns a value from a JSON object or array given its path.

This function provides “safe” navigation of a JSON value. If the input path is a valid path for the input JSON object/array, the JSON value at the end of that path is returned:

```
db> select json_get(to_json('{
...   "q": 1,
...   "w": [2, "foo"],
...   "e": true
... }'), 'w', '1');
{Json("\\"foo\\")}
```

This is useful when certain structure of JSON data is assumed, but cannot be reliably guaranteed. If the path cannot be followed for any reason, the empty set is returned:

```
db> select json_get(to_json('{
...   "q": 1,
...   "w": [2, "foo"],
...   "e": true
... }'), 'w', '2');
{}
```

If you want to supply your own default for the case where the path cannot be followed, you can do so using the `coalesce` operator:

```
db> select json_get(to_json('{
...   "q": 1,
...   "w": [2, "foo"],
...   "e": true
... }'), 'w', '2') ?? <json>'mydefault';
{Json("\\"mydefault\\")}
```

## function std::json\_set → optional json

Returns an updated JSON target with a new value.

```
db> select json_set(
...   to_json('{"a": 10, "b": 20}'),
...   'a',
...   value := <json>true,
... );
{Json("{\\\"a\\\": true, \\\"b\\\": 20}")}

db> select json_set(
...   to_json('{"a": {"b": {}}}'),
...   'a', 'b', 'c',
...   value := <json>42,
```

(continues on next page)

(continued from previous page)

```
... );
{Json("{\"a\": {\"b\": {\"c\": 42}}})}
```

If `create_if_missing` is set to `false`, a new path for the value won't be created:

```
db> select json_set(
...   to_json('{"a": 10, "b": 20}),
...   '',
...   value := <json>42,
... );
{Json("{\"a\": 10, \"b\": 20, \"\": 42"})}
db> select json_set(
...   to_json('{"a": 10, "b": 20}),
...   '',
...   value := <json>42,
...   create_if_missing := false,
... );
{Json("{\"a\": 10, \"b\": 20})}
```

The `empty_treatment` parameter defines the behavior of the function if an empty set is passed as `new_value`. This parameter can take these values:

- `ReturnEmpty`: return empty set, default
- `ReturnTarget`: return `target` unmodified
- `Error`: raise an `InvalidValueError`
- `UseNull`: use a `null` JSON value
- `DeleteKey`: delete the object key

```
db> select json_set(
...   to_json('{"a": 10, "b": 20}),
...   'a',
...   value := <json>{}
... );
{}
db> select json_set(
...   to_json('{"a": 10, "b": 20}),
...   'a',
...   value := <json>{},
...   empty_treatment := JsonEmpty.ReturnTarget,
... );
{Json("{\"a\": 10, \"b\": 20"})}
db> select json_set(
...   to_json('{"a": 10, "b": 20}),
...   'a',
...   value := <json>{},
...   empty_treatment := JsonEmpty.Error,
... );
InvalidValueError: invalid empty JSON value
db> select json_set(
...   to_json('{"a": 10, "b": 20}),
...   'a',
```

(continues on next page)

(continued from previous page)

```
...   value := <json>{},
...   empty_treatment := JsonEmpty.UseNull,
... );
{Json("{\"a\": null, \"b\": 20}")}

db> select json_set(
...   to_json('{"a": 10, "b": 20}'),
...   'a',
...   value := <json>{},
...   empty_treatment := JsonEmpty.DeleteKey,
... );
{Json("{\"b\": 20}")}
```

**function std::json\_object\_pack → json**

Returns the given set of key/value tuples as a JSON object.

```
db> select json_object_pack({
...   ("foo", to_json("1")),
...   ("bar", to_json("null")),
...   ("baz", to_json("[]"))
... });
{Json("{\"bar\": null, \"baz\": [], \"foo\": 1}")}
```

If the key/value tuples being packed have common keys, the last value for each key will make the final object.

```
db> select json_object_pack({
...   ("hello", <json>"world"),
...   ("hello", <json>true)
... });
{Json("{\"hello\": true}")}
```

**function std::json\_object\_unpack → set of tuple<str, json>**

Returns the data in a JSON object as a set of key/value tuples.

Calling this function on anything other than a JSON object will result in a runtime error.

```
db> select json_object_unpack(to_json('{
...   "q": 1,
...   "w": [2, "foo"],
...   "e": true
... }'));
{('e', Json("true")), ('q', Json("1")), ('w', Json("[2, \"foo\"]"))}
```

**function std::json\_typeof → str****Index Keywords** type

Returns the type of the outermost JSON value as a string.

Possible return values are: 'object', 'array', 'string', 'number', 'boolean', or 'null':

```
db> select json_typeof(<json>2);
{'number'}
db> select json_typeof(to_json('null'));
{'null'}
db> select json_typeof(to_json('{"a": 2}'));
{'object'}
```

## 5.9 UUIDs

<code>uuid</code>	UUID type
<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>uuid_generate_v1mc()</code>	Return a version 1 UUID.
<code>uuid_generate_v4()</code>	Return a version 4 UUID.

### `type uuid`

Universally Unique Identifiers (UUID).

For formal definition see RFC 4122 and ISO/IEC 9834-8:2005.

Every `Object` has a globally unique property `id` represented by a UUID value.

A UUID can be cast to an object type if an object of that type with a matching ID exists.

```
db> select <Hero><uuid>'01d9cc22-b776-11ed-8bef-73f84c7e91e7';
{default::Hero {id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7}}
```

### `function std::uuid_generate_v1mc → uuid`

Return a version 1 UUID.

The algorithm uses a random multicast MAC address instead of the real MAC address of the computer.

The UUID will contain 47 random bits, 60 bits representing the current time, and 14 bits of clock sequence that may be used to ensure uniqueness. The rest of the bits indicate the version of the UUID.

This is the default function used to populate the `id` column.

```
db> select uuid_generate_v1mc();
{1893e2b6-57ce-11e8-8005-13d4be166783}
```

### `function std::uuid_generate_v4 → uuid`

Return a version 4 UUID.

The UUID is derived entirely from random numbers: it will contain 122 random bits and 6 version bits.

It is permitted to override the `default` of the `id` column with a call to this function, but this should be done with caution: fully random ids will be less clustered than time-based id, which may lead to worse index performance.

```
db> select uuid_generate_v4();
{92673afc-9c4f-42b3-8273-afe0053f0f48}
```

## 5.10 Enums

**edb-alt-title** Enum Type

**type enum**

**Index Keywords** enum

An enumerated type is a data type consisting of an ordered list of values.

An enum type can be declared in a schema by using the following syntax:

```
scalar type Color extending enum<Red, Green, Blue>;
```

Enum values can then be accessed directly:

```
db> select Color.Red is Color;
{true}
```

*Casting* can be used to obtain an enum value in an expression:

```
db> select 'Red' is Color;
{false}
db> select <Color>'Red' is Color;
{true}
db> select <Color>'Red' = Color.Red;
{true}
```

---

**Note:** The enum values in EdgeQL are string-like in the fact that they can contain any characters that the strings can. This is different from some other languages where enum values are identifier-like and thus cannot contain some characters. For example, when working with GraphQL enum values that contain characters that aren't allowed in identifiers cannot be properly reflected. To address this, consider using only identifier-like enum values in cases where such compatibility is needed.

## 5.11 Dates and Times

**edb-alt-title** Types, Functions, and Operators for Dates and Times

<code>datetime</code>	Timezone-aware point in time
<code>duration</code>	Absolute time span
<code>cal::local_datetime</code>	Date and time w/o timezone
<code>cal::local_date</code>	Date type
<code>cal::local_time</code>	Time type
<code>cal::relative_duration</code>	Relative time span
<code>cal::date_duration</code>	Relative time span in days
<code>dt + dt</code>	Adds a duration and any other datetime value.
<code>dt - dt</code>	Subtracts two compatible datetime or duration values.
<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>to_str()</code>	Render a date/time value to a string.
<code>to_datetime()</code>	Create a datetime value.
<code>cal::to_local_datetime()</code>	Create a cal::local_datetime value.
<code>cal::to_local_date()</code>	Create a cal::local_date value.
<code>cal::to_local_time()</code>	Create a cal::local_time value.
<code>to_duration()</code>	Create a duration value.
<code>cal::to_relative_duration()</code>	Create a cal::relative_duration value.
<code>cal::to_date_duration()</code>	Create a cal::date_duration value.
<code>datetime_get()</code>	Returns the element of a date/time given a unit name.
<code>cal::time_get()</code>	Returns the element of a time value given a unit name.
<code>cal::date_get()</code>	Returns the element of a date given a unit name.
<code>duration_get()</code>	Returns the element of a duration given a unit name.
<code>datetime_truncate()</code>	Truncates the input datetime to a particular precision.
<code>duration_truncate()</code>	Truncates the input duration to a particular precision.
<code>datetime_current()</code>	Returns the server's current date and time.
<code>datetime_of_transaction()</code>	Returns the date and time of the start of the current transaction.
<code>datetime_of_statement()</code>	Returns the date and time of the start of the current statement.
<code>cal::duration_normalize_hours()</code>	Convert 24-hour chunks into days.
<code>cal::duration_normalize_days()</code>	Convert 30-day chunks into months.

EdgeDB offers two ways of representing date/time values:

- a timezone-aware `std::datetime` type;
- a set of “local” date/time types, not attached to any particular timezone: `cal::local_datetime`, `cal::local_date`, and `cal::local_time`.

There are also two different ways of measuring duration:

- `duration` for using absolute and unambiguous units;
- `cal::relative_duration` for using fuzzy units like years, months and days in addition to the absolute units.

All related operators, functions, and type casts are designed to maintain a strict separation between timezone-aware and “local” date/time values.

EdgeDB stores and outputs timezone-aware values in UTC format.

---

**Note:** All date/time types are restricted to years between 1 and 9999, including the years 1 and 9999.

Although many systems support ISO 8601 date/time formatting in theory, in practice the formatting before year 1 and after 9999 tends to be inconsistent. As such, dates outside this range are not reliably portable.

---

**type `datetime`**

Represents a timezone-aware moment in time.

All dates must correspond to dates that exist in the proleptic Gregorian calendar.

*Casting* is a simple way to obtain a `datetime` value in an expression:

```
select <datetime>'2018-05-07T15:01:22.306916+00';
select <datetime>'2018-05-07T15:01:22+00';
```

When casting `datetime` from strings, the string must follow the ISO 8601 format with a timezone included.

```
db> select <datetime>'January 01 2019 UTC';
InvalidValueError: invalid input syntax for type
std::datetime: 'January 01 2019 UTC'
Hint: Please use ISO8601 format. Alternatively "to_datetime"
function provides custom formatting options.

db> select <datetime>'2019-01-01T15:01:22';
InvalidValueError: invalid input syntax for type
std::datetime: '2019-01-01T15:01:22'
Hint: Please use ISO8601 format. Alternatively "to_datetime"
function provides custom formatting options.
```

All `datetime` values are restricted to the range from year 1 to 9999.

For more information regarding interacting with this type, see `datetime_get()`, `to_datetime()`, and `to_str()`.

**type `cal::local_datetime`**

A type for representing a date and time without a timezone.

*Casting* is a simple way to obtain a `cal::local_datetime` value in an expression:

```
select <cal::local_datetime>'2018-05-07T15:01:22.306916';
select <cal::local_datetime>'2018-05-07T15:01:22';
```

When casting `cal::local_datetime` from strings, the string must follow the ISO 8601 format without timezone:

```
db> select <cal::local_datetime>'2019-01-01T15:01:22+00';
InvalidValueError: invalid input syntax for type
cal::local_datetime: '2019-01-01T15:01:22+00'
Hint: Please use ISO8601 format. Alternatively
"cal::to_local_datetime" function provides custom formatting
options.

db> select <cal::local_datetime>'January 01 2019';
InvalidValueError: invalid input syntax for type
cal::local_datetime: 'January 01 2019'
Hint: Please use ISO8601 format. Alternatively
"cal::to_local_datetime" function provides custom formatting
options.
```

All `datetime` values are restricted to the range from year 1 to 9999.

For more information regarding interacting with this type, see `datetime_get()`, `cal::to_local_datetime()`, and `to_str()`.

---

#### **type cal::local\_date**

A type for representing a date without a timezone.

*Casting* is a simple way to obtain a `cal::local_date` value in an expression:

```
select <cal::local_date>'2018-05-07';
```

When casting `cal::local_date` from strings, the string must follow the ISO 8601 date format.

For more information regarding interacting with this type, see `cal::date_get()`, `cal::to_local_date()`, and `to_str()`.

---

#### **type cal::local\_time**

A type for representing a time without a timezone.

*Casting* is a simple way to obtain a `cal::local_time` value in an expression:

```
select <cal::local_time>'15:01:22.306916';
select <cal::local_time>'15:01:22';
```

When casting `cal::local_time` from strings, the string must follow the ISO 8601 time format.

For more information regarding interacting with this type, see `cal::time_get()`, `cal::to_local_time()`, and `to_str()`.

---

#### **type duration**

A type for representing a span of time.

A `duration` is a fixed number of seconds and microseconds and isn't adjusted by timezone, length of month, or anything else in datetime calculations.

When converting from a string, only units of 'microseconds', 'milliseconds', 'seconds', 'minutes', and 'hours' are valid:

```
db> select <duration>'45.6 seconds';
{<duration>'0:00:45.6'}
db> select <duration>'15 milliseconds';
{<duration>'0:00:00.015'}
db> select <duration>'48 hours 45 minutes';
{<duration>'48:45:00'}
db> select <duration>'11 months';
edgedb error: InvalidValueError: invalid input syntax for type
std::duration: '11 months'
Hint: Units bigger than hours cannot be used for std::duration.
```

All date/time types support the + and - arithmetic operations with durations:

```
db> select <datetime>'2019-01-01T00:00:00Z' - <duration>'24 hours';
{<datetime>'2018-12-31T00:00:00+00:00'}
db> select <cal::local_time>'22:00' + <duration>'1 hour';
{<cal::local_time>'23:00:00'}
```

For more information regarding interacting with this type, see `to_duration()`, and `to_str()` and date/time operators.

### `type cal::relative_duration`

A type for representing a relative span of time.

Unlike `std::duration`, `cal::relative_duration` is an imprecise form of measurement. When months and days are used, the same relative duration could have a different absolute duration depending on the date you're measuring from.

For example 2020 was a leap year and had 366 days. Notice how the number of hours in each year below is different:

```
db> with
...     first_day_of_2020 := <datetime>'2020-01-01T00:00:00Z',
...     one_year := <cal::relative_duration>'1 year',
...     first_day_of_next_year := first_day_of_2020 + one_year
... select first_day_of_next_year - first_day_of_2020;
{<duration>'8784:00:00'}
db> with
...     first_day_of_2019 := <datetime>'2019-01-01T00:00:00Z',
...     one_year := <cal::relative_duration>'1 year',
...     first_day_of_next_year := first_day_of_2019 + one_year
... select first_day_of_next_year - first_day_of_2019;
{<duration>'8760:00:00'}
```

When converting from a string, only the following units are valid:

- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'
- 'decades'
- 'centuries'
- 'millennia'

Examples of units usage:

```
select <cal::relative_duration>'45.6 seconds';
select <cal::relative_duration>'15 milliseconds';
select <cal::relative_duration>'3 weeks 45 minutes';
select <cal::relative_duration>'-7 millennia';
```

All date/time types support the + and - arithmetic operations with `relative_duration`:

```
db> select <datetime>'2019-01-01T00:00:00Z' -
...      <cal::relative_duration>'3 years';
{<datetime>'2016-01-01T00:00:00+00:00'}
db> select <cal::local_time>'22:00' +
...      <cal::relative_duration>'1 hour';
{<cal::local_time>'23:00:00'}
```

If an arithmetic operation results in a day that doesn't exist in the given month, the last day of the month will be used instead:

```
db> select <cal::local_datetime>"2021-01-31T15:00:00" +
...      <cal::relative_duration>"1 month";
{<cal::local_datetime>'2021-02-28T15:00:00'}
```

For arithmetic operations involving a `cal::relative_duration` consisting of multiple components (units), higher-order components are applied first followed by lower-order components.

```
db> select <cal::local_datetime>"2021-04-30T15:00:00" +
...      <cal::relative_duration>"1 month 1 day";
{<cal::local_datetime>'2021-05-31T15:00:00'}
```

If you add the same components split into separate durations, adding the higher-order units first followed by the lower-order units, the calculation produces the same result as in the previous example:

```
db> select <cal::local_datetime>"2021-04-30T15:00:00" +
...      <cal::relative_duration>"1 month" +
...      <cal::relative_duration>"1 day";
{<cal::local_datetime>'2021-05-31T15:00:00'}
```

When the order of operations is reversed, the result may be different for some corner cases:

```
db> select <cal::local_datetime>"2021-04-30T15:00:00" +
...      <cal::relative_duration>"1 day" +
...      <cal::relative_duration>"1 month";
{<cal::local_datetime>'2021-06-01T15:00:00'}
```

## Gotchas

Due to the implementation of `relative_duration` logic, arithmetic operations may behave counterintuitively.

### Non-associative

```
db> select <cal::local_datetime>'2021-01-31T00:00:00' +
...      <cal::relative_duration>'1 month' +
...      <cal::relative_duration>'1 month';
{<cal::local_datetime>'2021-03-28T00:00:00'}
db> select <cal::local_datetime>'2021-01-31T00:00:00' +
...      (<cal::relative_duration>'1 month' +
...      <cal::relative_duration>'1 month');
{<cal::local_datetime>'2021-03-31T00:00:00'}
```

### Lossy

```
db> with m := <cal::relative_duration>'1 month'
... select <cal::local_date>'2021-01-31' + m
...
...      =
...      <cal::local_date>'2021-01-30' + m;
{true}
```

**Asymmetric**

```
db> with m := <cal::relative_duration>'1 month'
... select <cal::local_date>'2021-01-31' + m - m;
{<cal::local_date>'2021-01-28'}
```

**Non-monotonic**

```
db> with m := <cal::relative_duration>'1 month'
... select <cal::local_datetime>'2021-01-31T01:00:00' + m
...
...      <
...      <cal::local_datetime>'2021-01-30T23:00:00' + m;
{true}
db> with m := <cal::relative_duration>'2 month'
... select <cal::local_datetime>'2021-01-31T01:00:00' + m
...
...      <
...      <cal::local_datetime>'2021-01-30T23:00:00' + m;
{false}
```

For more information regarding interacting with this type, see [cal::to\\_relative\\_duration\(\)](#), and [to\\_str\(\)](#) and date/time [operators](#).

**type cal::date\_duration**

A type for representing a span of time in days.

This type is similar to [cal::relative\\_duration](#), except it only uses 2 units: months and days. It is the result of subtracting one [cal::local\\_date](#) from another. The purpose of this type is to allow performing + and - operations on a [cal::local\\_date](#) and to produce a [cal::local\\_date](#) as the result:

```
db> select <cal::local_date>'2022-06-30' -
...     <cal::local_date>'2022-06-25';
{<cal::date_duration>'P5D'}
db> select <cal::local_date>'2022-06-25' +
...     <cal::date_duration>'5 days';
{<cal::local_date>'2022-06-30'}
db> select <cal::local_date>'2022-06-25' -
...     <cal::date_duration>'5 days';
{<cal::local_date>'2022-06-20'}
```

When converting from a string, only the following units are valid:

- 'days',
- 'weeks',
- 'months',
- 'years',
- 'decades',

- 'centuries',
- 'millennia'.

```
select <cal::date_duration>'45 days';
select <cal::date_duration>'3 weeks 5 days';
select <cal::date_duration>'-7 millennia';
```

In most cases, `date_duration` is fully compatible with `cal::relative_duration` and shares the same general behavior and caveats. EdgeDB will apply type coercion in the event it expects a `cal::relative_duration` and finds a `cal::date_duration` instead.

For more information regarding interacting with this type, see [cal::to\\_date\\_duration\(\)](#) and date/time operators.

---

```
operator datetime + duration -> datetime
operator
datetime + cal::relative_duration -> cal::relative_duration
operator duration + duration -> duration
operator
duration + cal::relative_duration -> cal::relative_duration
operator
cal::relative_duration + cal::relative_duration -> cal::relative_duration
operator
cal::local_datetime + cal::relative_duration -> cal::relative_duration
operator cal::local_datetime + duration -> cal::local_datetime
operator
cal::local_time + cal::relative_duration -> cal::relative_duration
operator cal::local_time + duration -> cal::local_time
operator cal::local_date + cal::date_duration -> cal::local_date
operator
cal::date_duration + cal::date_duration -> cal::date_duration
operator
cal::local_date + cal::relative_duration -> cal::local_datetime
operator cal::local_date + duration -> cal::local_datetime
```

Adds a duration and any other datetime value.

This operator is commutative.

```
db> select <cal::local_time>'22:00' + <duration>'1 hour';
{<cal::local_time>'23:00:00'}
db> select <duration>'1 hour' + <cal::local_time>'22:00';
{<cal::local_time>'23:00:00'}
db> select <duration>'1 hour' + <duration>'2 hours';
{10800s}
```

---

```
operator duration - duration -> duration
operator datetime - datetime -> duration
operator datetime - duration -> datetime
operator datetime - cal::relative_duration -> datetime
operator
cal::relative_duration - cal::relative_duration -> cal::relative_duration
operator
cal::local_datetime - cal::local_datetime -> cal::relative_duration
```

```

operator
cal::local_datetime - cal::relative_duration           -> cal::local_datetime
operator cal::local_datetime - duration                -> cal::local_datetime
operator
cal::local_time - cal::local_time                    -> cal::relative_duration
operator
cal::local_time - cal::relative_duration            -> cal::local_time
operator cal::local_time - duration -> cal::local_time
operator
cal::date_duration - cal::date_duration            -> cal::date_duration
operator
cal::local_date - cal::local_date                  -> cal::date_duration
operator
cal::local_date - cal::date_duration              -> cal::local_date
operator
cal::local_date - cal::relative_duration          -> cal::local_datetime
operator cal::local_date - duration -> cal::local_datetime
operator
duration - cal::relative_duration                 -> cal::relative_duration
operator
cal::relative_duration - duration                 -> cal::relative_duration

```

Subtracts two compatible datetime or duration values.

```

db> select <datetime>'2019-01-01T01:02:03+00' -
...   <duration>'24 hours';
{<datetime>'2018-12-31T01:02:03Z'}
db> select <datetime>'2019-01-01T01:02:03+00' -
...   <datetime>'2019-02-01T01:02:03+00';
{-2678400s}
db> select <duration>'1 hour' -
...   <duration>'2 hours';
{-3600s}

```

When subtracting a `cal::local_date` type from another, the result is given as a whole number of days using the `cal::date_duration` type:

```

db> select <cal::local_date>'2022-06-25' -
...   <cal::local_date>'2019-02-01';
{<cal::date_duration>'P1240D'}

```

---

**Note:** Subtraction doesn't make sense for some type combinations. You couldn't subtract a point in time from a duration, so neither can EdgeDB (although the inverse — subtracting a duration from a point in time — is perfectly fine). You also couldn't subtract a timezone-aware datetime from a local one or vice versa. If you attempt any of these, EdgeDB will raise an exception as shown in these examples.

When subtracting a date/time object from a time interval, an exception will be raised:

```

db> select <duration>'1 day' -
...   <datetime>'2019-01-01T01:02:03+00';
QueryError: operator '-' cannot be applied to operands ...

```

An exception will also be raised when trying to subtract a timezone-aware `std::datetime` type from `cal::local_datetime` or vice versa:

```
db> select <datetime>'2019-01-01T01:02:03+00' -
...   <cal::local_datetime>'2019-02-01T01:02:03';
QueryError: operator '-' cannot be applied to operands...
db> select <cal::local_datetime>'2019-02-01T01:02:03' -
...   <datetime>'2019-01-01T01:02:03+00';
QueryError: operator '-' cannot be applied to operands...
```

---

**function std::datetime\_current** → datetime**Index Keywords** now

Returns the server's current date and time.

```
db> select datetime_current();
{<datetime>'2018-05-14T20:07:11.755827Z'}
```

This function is volatile since it always returns the current time when it is called. As a result, it cannot be used in *computed properties defined in schema*. This does *not* apply to computed properties outside of schema.

---

**function std::datetime\_of\_transaction** → datetime**Index Keywords** now

Returns the date and time of the start of the current transaction.

This function is non-volatile since it returns the current time when the transaction is started, not when the function is called. As a result, it can be used in *computed properties* defined in schema.

---

**function std::datetime\_of\_statement** → datetime**Index Keywords** now

Returns the date and time of the start of the current statement.

This function is non-volatile since it returns the current time when the statement is started, not when the function is called. As a result, it can be used in *computed properties* defined in schema.

---

**function std::datetime\_get** → float64**function std::datetime\_get** → float64

Returns the element of a date/time given a unit name.

You may pass any of these unit names for *el*:

- 'epochseconds' - the number of seconds since 1970-01-01 00:00:00 UTC (Unix epoch) for *datetime* or local time for *cal::local\_datetime*. It can be negative.
- 'century' - the century according to the Gregorian calendar
- 'day' - the day of the month (1-31)
- 'decade' - the decade (year divided by 10 and rounded down)
- 'dow' - the day of the week from Sunday (0) to Saturday (6)
- 'doy' - the day of the year (1-366)

- 'hour' - the hour (0-23)
- 'isodow' - the ISO day of the week from Monday (1) to Sunday (7)
- 'isoyear' - the ISO 8601 week-numbering year that the date falls in. See the 'week' element for more details.
- 'microseconds' - the seconds including fractional value expressed as microseconds
- 'millennium' - the millennium. The third millennium started on Jan 1, 2001.
- 'milliseconds' - the seconds including fractional value expressed as milliseconds
- 'minutes' - the minutes (0-59)
- 'month' - the month of the year (1-12)
- 'quarter' - the quarter of the year (1-4)
- 'seconds' - the seconds, including fractional value from 0 up to and not including 60
- 'week' - the number of the ISO 8601 week-numbering week of the year. ISO weeks are defined to start on Mondays and the first week of a year must contain Jan 4 of that year.
- 'year' - the year

```
db> select datetime_get(
...     <datetime>'2018-05-07T15:01:22.306916+00',
...     'epochseconds');
{1525705282.306916}

db> select datetime_get(
...     <datetime>'2018-05-07T15:01:22.306916+00',
...     'year');
{2018}

db> select datetime_get(
...     <datetime>'2018-05-07T15:01:22.306916+00',
...     'quarter');
{2}

db> select datetime_get(
...     <datetime>'2018-05-07T15:01:22.306916+00',
...     'doy');
{127}

db> select datetime_get(
...     <datetime>'2018-05-07T15:01:22.306916+00',
...     'hour');
{15}
```

**function cal::time\_get → float64**

Returns the element of a time value given a unit name.

You may pass any of these unit names for *el*:

- 'midnightseconds'
- 'hour'

- 'microseconds'
- 'milliseconds'
- 'minutes'
- 'seconds'

For full description of what these elements extract see [datetime\\_get\(\)](#).

```
db> select cal::time_get(  
...     <cal::local_time>'15:01:22.306916', 'minutes');  
{1}  
  
db> select cal::time_get(  
...     <cal::local_time>'15:01:22.306916', 'milliseconds');  
{22306.916}
```

---

**function cal::date\_get** → float64

Returns the element of a date given a unit name.

The `cal::local_date` scalar has the following elements available for extraction:

- 'century' - the century according to the Gregorian calendar
- 'day' - the day of the month (1-31)
- 'decade' - the decade (year divided by 10 and rounded down)
- 'dow' - the day of the week from Sunday (0) to Saturday (6)
- 'doy' - the day of the year (1-366)
- 'isodow' - the ISO day of the week from Monday (1) to Sunday (7)
- 'isoyear' - the ISO 8601 week-numbering year that the date falls in. See the 'week' element for more details.
- 'millennium' - the millennium. The third millennium started on Jan 1, 2001.
- 'month' - the month of the year (1-12)
- 'quarter' - the quarter of the year (1-4) not including 60
- 'week' - the number of the ISO 8601 week-numbering week of the year. ISO weeks are defined to start on Mondays and the first week of a year must contain Jan 4 of that year.
- 'year' - the year

```
db> select cal::date_get(  
...     <cal::local_date>'2018-05-07', 'century');  
{21}  
  
db> select cal::date_get(  
...     <cal::local_date>'2018-05-07', 'year');  
{2018}  
  
db> select cal::date_get(  
...     <cal::local_date>'2018-05-07', 'month');  
{5}
```

(continues on next page)

(continued from previous page)

```
db> select cal::date_get(
...     <cal::local_date>'2018-05-07', 'doy');
{127}
```

```
function std::duration_get → float64
function std::duration_get → float64
function std::duration_get → float64
```

Returns the element of a duration given a unit name.

You may pass any of these unit names as `e1`:

- 'millennium' - number of 1000-year chunks rounded down
- 'century' - number of centuries rounded down
- 'decade' - number of decades rounded down
- 'year' - number of years rounded down
- 'quarter' - remaining quarters after whole years are accounted for
- 'month' - number of months left over after whole years are accounted for
- 'day' - number of days recorded in the duration
- 'hour' - number of hours
- 'minutes' - remaining minutes after whole hours are accounted for
- 'seconds' - remaining seconds, including fractional value after whole minutes are accounted for
- 'milliseconds' - remaining seconds including fractional value expressed as milliseconds
- 'microseconds' - remaining seconds including fractional value expressed as microseconds

**Note:** Only for units 'month' or larger or for units 'hour' or smaller will you receive a total across multiple units expressed in the original duration. See *Gotchas* below for details.

Additionally, it's possible to convert a given duration into seconds:

- 'totalseconds' - the number of seconds represented by the duration. It will be approximate for `cal::relative_duration` and `cal::date_duration` for units 'month' or larger because a month is assumed to be 30 days exactly.

The `duration` scalar has only 'hour' and smaller units available for extraction.

The `cal::relative_duration` scalar has all of the units available for extraction.

The `cal::date_duration` scalar only has 'date' and larger units available for extraction.

```
db> select duration_get(
...     <cal::relative_duration>'400 months', 'year');
{33}
db> select duration_get(
...     <cal::date_duration>'400 months', 'month');
{4}
db> select duration_get(
```

(continues on next page)

(continued from previous page)

```

...   <cal::relative_duration>'1 month 20 days 30 hours',
...   'day');
{20}
db> select duration_get(
...   <cal::relative_duration>'30 hours', 'hour');
{30}
db> select duration_get(
...   <cal::relative_duration>'1 month 20 days 30 hours',
...   'hour');
{30}
db> select duration_get(<duration>'30 hours', 'hour');
{30}
db> select duration_get(
...   <cal::relative_duration>'1 month 20 days 30 hours',
...   'totalseconds');
{4428000}
db> select duration_get(
...   <duration>'30 hours', 'totalseconds');
{108000}

```

## Gotchas

This function will provide you with a calculated total for the unit passed as `e1`, but only within the given “size class” of the unit. These size classes exist because they are logical breakpoints that we can’t reliably convert values across. A month might be 30 days long, or it might be 28 or 29 or 31. A day is generally 24 hours, but with daylight savings, it might be longer or shorter.

As a result, it’s impossible to convert across these lines in a way that works in every situation. For some use cases, assuming a 30 day month works fine. For others, it might not. The size classes are as follows:

- 'month' and larger
- 'day'
- 'hour' and smaller

For example, if you specify 'day' as your `e1` argument, the function will return only the number of days expressed as N days in your duration. It will not add another day to the returned count for every 24 hours (defined as 24 hours) in the duration, nor will it consider the months’ constituent day counts in the returned value. Specifying 'decade' for `e1` will total up all decades represented in units 'month' and larger, but it will not add a decade’s worth of days to the returned value as an additional decade.

In this example, the duration represents more than a day’s time, but since 'day' and 'hour' are in different size classes, the extra day stemming from the duration’s hours is not added.

```

db> select duration_get(
...   <cal::relative_duration>'1 day 36 hours', 'day');
{1}

```

In this counter example, both the decades and months are pooled together since they are in the same size class. The return value is 5: the 2 'decades' and the 3 decades in '400 months'.

```
db> select duration_get(
...     <cal::relative_duration>'2 decades 400 months', 'decade');
{5}
```

If a unit from a smaller size class would contribute to your desired unit's total, it is not added.

```
db> select duration_get(
...     <cal::relative_duration>'1 year 400 days', 'year');
{1}
```

When you request a unit in the smallest size class, it will be pooled with other durations in the same size class.

```
db> select duration_get(
...     <cal::relative_duration>'20 hours 3600 seconds', 'hour');
{21}
```

Seconds and smaller units always return remaining time in that unit after accounting for the next larger unit.

```
db> select duration_get(
...     <cal::relative_duration>'20 hours 3600 seconds', 'seconds');
{0}
db> select duration_get(
...     <cal::relative_duration>'20 hours 3630 seconds', 'seconds');
{30}
```

Normalization and truncation may help you deal with this. If your use case allows for making assumptions about the duration of a month or a day, you can make those conversions for yourself using the `cal::duration_normalize_hours()` or `cal::duration_normalize_days()` functions. If you got back a duration as a result of a datetime calculation and don't need the level of granularity you have, you can truncate the value with `duration_truncate()`.

### **function std::datetime\_truncate** → datetime

Truncates the input datetime to a particular precision.

The valid units in order of decreasing precision are:

- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'quarters'
- 'years'
- 'decades'
- 'centuries'

```
db> select datetime_truncate(
...     <datetime>'2018-05-07T15:01:22.306916+00', 'years');
{<datetime>'2018-01-01T00:00:00Z'}

db> select datetime_truncate(
...     <datetime>'2018-05-07T15:01:22.306916+00', 'quarters');
{<datetime>'2018-04-01T00:00:00Z'}

db> select datetime_truncate(
...     <datetime>'2018-05-07T15:01:22.306916+00', 'days');
{<datetime>'2018-05-07T00:00:00Z'}

db> select datetime_truncate(
...     <datetime>'2018-05-07T15:01:22.306916+00', 'hours');
{<datetime>'2018-05-07T15:00:00Z'}
```

---

```
function std::duration_truncate → duration
function std::duration_truncate → cal::relative_duration
```

Truncates the input duration to a particular precision.

The valid units for *duration* are:

- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'

In addition to the above the following are also valid for *cal::relative\_duration*:

- 'days'
- 'weeks'
- 'months'
- 'years'
- 'decades'
- 'centuries'

```
db> select duration_truncate(
...     <duration>'15:01:22', 'hours');
{<duration>'15:00:00'}
db> select duration_truncate(
...     <duration>'15:01:22.306916', 'minutes');
{<duration>'15:01:00'}
db> select duration_truncate(
...     <cal::relative_duration>'400 months', 'years');
{<cal::relative_duration>'P33Y'}
db> select duration_truncate(
...     <cal::relative_duration>'400 months', 'decades');
{<cal::relative_duration>'P30Y'}
```

---

```
function std::to_datetime → datetime
```

**Index Keywords** parse datetime

Create a `datetime` value.

The `datetime` value can be parsed from the input `str s`. By default, the input is expected to conform to ISO 8601 format. However, the optional argument `fmt` can be used to override the `input format` to other forms.

```
db> select to_datetime('2018-05-07T15:01:22.306916+00');
{<datetime>'2018-05-07T15:01:22.306916Z'}
db> select to_datetime('2018-05-07T15:01:22+00');
{<datetime>'2018-05-07T15:01:22Z'}
db> select to_datetime('May 7th, 2018 15:01:22 +00',
...                      'Mon DDth, YYYY HH24:MI:SS TZH');
{<datetime>'2018-05-07T15:01:22Z'}
```

Alternatively, the `datetime` value can be constructed from a `cal::local_datetime` value:

```
db> select to_datetime(
...     <cal::local_datetime>'2019-01-01T01:02:03', 'HKT');
{<datetime>'2018-12-31T17:02:03Z'}
```

Another way to construct a the `datetime` value is to specify it in terms of its component parts: `year`, `month`, `day`, `hour`, `min`, `sec`, and `timezone`.

```
db> select to_datetime(
...     2018, 5, 7, 15, 1, 22.306916, 'UTC');
{<datetime>'2018-05-07T15:01:22.306916000Z'}
```

Finally, it is also possible to convert a Unix timestamp to a `datetime`

```
db> select to_datetime(1590595184.584);
{<datetime>'2020-05-27T15:59:44.584000000Z'}
```

---

```
function cal::to_local_datetime → local_datetime
function cal::to_local_datetime → local_datetime
function cal::to_local_datetime → local_datetime
```

**Index Keywords** parse local\_datetime

Create a `cal::local_datetime` value.

Similar to `to_datetime()`, the `cal::local_datetime` value can be parsed from the input `str s` with an optional `fmt` argument or it can be given in terms of its component parts: `year`, `month`, `day`, `hour`, `min`, `sec`.

For more details on formatting see [here](#).

```
db> select cal::to_local_datetime('2018-05-07T15:01:22.306916');
{<cal::local_datetime>'2018-05-07T15:01:22.306916'}
db> select cal::to_local_datetime('May 7th, 2018 15:01:22',
...                                'Mon DDth, YYYY HH24:MI:SS');
{<cal::local_datetime>'2018-05-07T15:01:22'}
db> select cal::to_local_datetime(
...      2018, 5, 7, 15, 1, 22.306916);
{<cal::local_datetime>'2018-05-07T15:01:22.306916'}
```

A timezone-aware `datetime` type can be converted to local datetime in the specified timezone:

```
db> select cal::to_local_datetime(
...     <datetime>'2018-12-31T22:00:00+08',
...     'US/Central');
{<cal::local_datetime>'2018-12-31T08:00:00'}
```

---

```
function cal::to_local_date → cal::local_date
function cal::to_local_date → cal::local_date
function cal::to_local_date → cal::local_date
```

**Index Keywords** `parse local_date`

Create a `cal::local_date` value.

Similar to `to_datetime()`, the `cal::local_date` value can be parsed from the input `str s` with an optional `fmt` argument or it can be given in terms of its component parts: `year, month, day`.

For more details on formatting see [here](#).

```
db> select cal::to_local_date('2018-05-07');
{<cal::local_date>'2018-05-07'}
db> select cal::to_local_date('May 7th, 2018', 'Mon DDth, YYYY');
{<cal::local_date>'2018-05-07'}
db> select cal::to_local_date(2018, 5, 7);
{<cal::local_date>'2018-05-07'}
```

A timezone-aware `datetime` type can be converted to local date in the specified timezone:

```
db> select cal::to_local_date(
...     <datetime>'2018-12-31T22:00:00+08',
...     'US/Central');
{<cal::local_date>'2019-01-01'}
```

---

```
function cal::to_local_time → local_time
function cal::to_local_time → local_time
function cal::to_local_time → local_time
```

**Index Keywords** `parse local_time`

Create a `cal::local_time` value.

Similar to `to_datetime()`, the `cal::local_time` value can be parsed from the input `str s` with an optional `fmt` argument or it can be given in terms of its component parts: `hour, min, sec`.

For more details on formatting see [here](#).

```
db> select cal::to_local_time('15:01:22.306916');
{<cal::local_time>'15:01:22.306916'}
db> select cal::to_local_time('03:01:22pm', 'HH:MI:SSam');
{<cal::local_time>'15:01:22'}
db> select cal::to_local_time(15, 1, 22.306916);
{<cal::local_time>'15:01:22.306916'}
```

A timezone-aware `datetime` type can be converted to local date in the specified timezone:

```
db> select cal::to_local_time(
...     <datetime>'2018-12-31T22:00:00+08',
...     'US/Pacific');
{<cal::local_time>'06:00:00'}
```

## **function std::to\_duration** → duration

**Index Keywords** duration

Create a `duration` value.

This function uses named `only` arguments to create a `duration` value. The available duration fields are: `hours`, `minutes`, `seconds`, `microseconds`.

```
db> select to_duration(hours := 1,
...                     minutes := 20,
...                     seconds := 45);
{4845s}
db> select to_duration(seconds := 4845);
{4845s}
```

## **function std::duration\_to\_seconds** → decimal

Return duration as total number of seconds in interval.

```
db> select duration_to_seconds(<duration>'1 hour');
{3600.000000n}
db> select duration_to_seconds(<duration>'10 second 123 ms');
{10.123000n}
```

## **function cal::to\_relative\_duration** → cal::relative\_duration

**Index Keywords** parse relative\_duration

Create a `cal::relative_duration` value.

This function uses named `only` arguments to create a `cal::relative_duration` value. The available duration fields are: `years`, `months`, `days`, `hours`, `minutes`, `seconds`, `microseconds`.

```
db> select cal::to_relative_duration(years := 5, minutes := 1);
{<cal::relative_duration>'P5YT1S'}
db> select cal::to_relative_duration(months := 3, days := 27);
{<cal::relative_duration>'P3M27D'}
```

**function** `cal::to_date_duration` → `cal::date_duration`

**Index Keywords** `parse date_duration`

Create a `cal::date_duration` value.

This function uses named only arguments to create a `cal::date_duration` value. The available duration fields are: `years`, `months`, `days`.

```
db> select cal::to_date_duration(years := 1, days := 3);
{<cal::date_duration>'P1Y3D'}
db> select cal::to_date_duration(days := 12);
{<cal::date_duration>'P12D'}
```

---

**function** `cal::duration_normalize_hours` → `cal::relative_duration`

**Index Keywords** `justify_hours`

Convert 24-hour chunks into days.

This function converts all 24-hour chunks into day units. The resulting `cal::relative_duration` is guaranteed to have less than 24 hours in total in the units smaller than days.

```
db> select cal::duration_normalize_hours(
...    <cal::relative_duration>'1312 hours');
{<cal::relative_duration>'P54DT16H'}
```

This is a lossless operation because 24 hours are always equal to 1 day in `cal::relative_duration` units.

This is sometimes used together with `cal::duration_normalize_days()`.

---

**function** `cal::duration_normalize_days` → `cal::relative_duration`

**function** `cal::duration_normalize_days` → `cal::date_duration`

**Index Keywords** `justify_days`

Convert 30-day chunks into months.

This function converts all 30-day chunks into month units. The resulting `cal::relative_duration` or `cal::date_duration` is guaranteed to have less than 30 day units.

```
db> select cal::duration_normalize_days(
...    <cal::relative_duration>'1312 days');
{<cal::relative_duration>'P3Y7M22D'}

db> select cal::duration_normalize_days(
...    <cal::date_duration>'1312 days');
{<cal::date_duration>'P3Y7M22D'}
```

This function is a form of approximation and does not preserve the exact duration.

This is often used together with `cal::duration_normalize_hours()`.

## 5.12 Arrays

### edb-alt-title Array Functions and Operators

<code>array[i]</code>	Accesses the array element at a given index.
<code>array[from:to]</code>	Produces a sub-array from an existing array.
<code>array ++ array</code>	Concatenates two arrays of the same type into one.
<code>= != ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>len()</code>	Returns the number of elements in the array.
<code>contains()</code>	Checks if an element is in the array.
<code>find()</code>	Finds the index of an element in the array.
<code>array_join()</code>	Renders an array to a string.
<code>array_fill()</code>	Returns an array of the specified size, filled with the provided value.
<code>array_replace()</code>	Returns an array with all occurrences of one value replaced by another.
<code>array_agg()</code>	Returns an array made from all of the input set elements.
<code>array_get()</code>	Returns the element of a given array at the specified index.
<code>array_unpack()</code>	Returns the elements of an array as a set.

Arrays store expressions of the *same type* in an ordered list.

### 5.12.1 Constructing arrays

An array constructor is an expression that consists of a sequence of comma-separated expressions *of the same type* enclosed in square brackets. It produces an array value:

```
"[" <expr> [, ...] "]"
```

For example:

```
db> select [1, 2, 3];
{[1, 2, 3]}
db> select [('a', 1), ('b', 2), ('c', 3)];
{[('a', 1), ('b', 2), ('c', 3)]}
```

### 5.12.2 Empty arrays

You can also create an empty array, but it must be done by providing the type information using type casting. EdgeDB cannot infer the type of an empty array created otherwise. For example:

```
db> select [];
QueryError: expression returns value of indeterminate type
Hint: Consider using an explicit type cast.
### select [];
###           ^
db> select <array<int64>>[];
{[]}
```

### 5.12.3 Reference

#### type array

**Index Keywords** array

An ordered list of values of the same type.

Array indexing starts at zero.

An array can contain any type except another array. In EdgeDB, arrays are always one-dimensional.

An array type is created implicitly when an *array constructor* is used:

```
db> select [1, 2];
{[1, 2]}
```

The array types themselves are denoted by array followed by their sub-type in angle brackets. These may appear in cast operations:

```
db> select <array<str>>[1, 4, 7];
{['1', '4', '7']}
db> select <array<bigint>>[1, 4, 7];
{[1n, 4n, 7n]}
```

Array types may also appear in schema declarations:

```
type Person {
    property str_array -> array<str>;
    property json_array -> array<json>;
}
```

```
type Person {
    str_array: array<str>;
    json_array: array<json>;
}
```

See also the list of standard *array functions*, as well as *generic functions* such as `len()`.

---

#### operator array<anytype> [ int64 ] -> anytype

Accesses the array element at a given index.

Example:

```
db> select [1, 2, 3][0];
{1}
db> select [(x := 1, y := 1), (x := 2, y := 3.3)][1];
{(x := 2, y := 3.3)}
```

This operator also allows accessing elements from the end of the array using a negative index:

```
db> select [1, 2, 3][-1];
{3}
```

Referencing a non-existent array element will result in an error:

```
db> select [1, 2, 3][4];
InvalidValueError: array index 4 is out of bounds
```

**operator array<anytype> [ int64 : int64 ] -> anytype**

Produces a sub-array from an existing array.

Omitting the lower bound of an array slice will default to a lower bound of zero.

Omitting the upper bound will default the upper bound to the length of the array.

The lower bound of an array slice is inclusive while the upper bound is not.

Examples:

```
db> select [1, 2, 3][0:2];
{[1, 2]}
db> select [1, 2, 3][2:];
{[3]}
db> select [1, 2, 3][:1];
{[1]}
db> select [1, 2, 3][-2:];
{[1]}
```

Referencing an array slice beyond the array boundaries will result in an empty array (unlike a direct reference to a specific index). Slicing with a lower bound less than the minimum index or a upper bound greater than the maximum index are functionally equivalent to not specifying those bounds for your slice:

```
db> select [1, 2, 3][1:20];
{[2, 3]}
db> select [1, 2, 3][10:20];
{[]}
```

**operator array<anytype> ++ array<anytype> -> array<anytype>**

Concatenates two arrays of the same type into one.

```
db> select [1, 2, 3] ++ [99, 98];
{[1, 2, 3, 99, 98]}
```

**function std::array\_agg → array<anytype>**

**Index Keywords** aggregate array set

Returns an array made from all of the input set elements.

The ordering of the input set will be preserved if specified:

```
db> select array_agg({2, 3, 5});
{[2, 3, 5]}

db> select array_agg(User.name order by User.name);
{['Alice', 'Bob', 'Joe', 'Sam']}
```

**function std::array\_get** → optional anytype**Index Keywords** array access get

Returns the element of a given *array* at the specified *index*.

If the index is out of the array's bounds, the *default* argument or {} (empty set) will be returned.

This works the same as the [array indexing operator](#), except that if the index is out of bounds, an empty set of the array element's type is returned instead of raising an exception:

```
db> select array_get([2, 3, 5], 1);
{3}
db> select array_get([2, 3, 5], 100);
{}
db> select array_get([2, 3, 5], 100, default := 42);
{42}
```

---

**function std::array\_unpack** → set of anytype**Index Keywords** set array unpack

Returns the elements of an array as a set.

---

**Note:** The ordering of the returned set is not guaranteed. However, if it is wrapped in a call to [enumerate\(\)](#), the assigned indexes are guaranteed to match the array.

```
db> select array_unpack([2, 3, 5]);
{3, 2, 5}
db> select enumerate(array_unpack([2, 3, 5]));
{(1, 3), (0, 2), (2, 5)}
```

---

**function std::array\_join** → str**Index Keywords** join array\_to\_string implode

Renders an array to a string.

Join a string array into a single string using a specified *delimiter*:

```
db> select to_str(['one', 'two', 'three'], ', ', '');
{'one, two, three'}
```

---

**function std::array\_fill** → array<anytype>**Index Keywords** fill

Returns an array of the specified size, filled with the provided value.

Create an array of size *n* where every element has the value *val*.

```
db> select array_fill(0, 5);
{[0, 0, 0, 0, 0]}
db> select array_fill('n/a', 3);
{['n/a', 'n/a', 'n/a']}
```

**function std::array\_replace** → array<anytype>

Returns an array with all occurrences of one value replaced by another.

Return an array where every *old* value is replaced with *new*.

```
db> select array_replace([1, 1, 2, 3, 5], 1, 99);
{[99, 99, 2, 3, 5]}
db> select array_replace(['h', 'e', 'l', 'l', 'o'], 'l', 'L');
{['h', 'e', 'L', 'L', 'o']}
```

## 5.13 Tuples

A tuple type is a heterogeneous sequence of other types. Tuples can be either *named* or *unnamed* (the default).

### 5.13.1 Constructing tuples

A tuple constructor is an expression that consists of a sequence of comma-separated expressions enclosed in parentheses. It produces a tuple value:

```
"(" <expr> [, ... ] ")"
```

Declare a *named tuple*:

```
"(" <identifier> := <expr> [, ... ] ")"
```

All elements in a named tuple must have a name.

A tuple constructor automatically creates a corresponding *tuple type*.

### 5.13.2 Accessing elements

An element of a tuple can be referenced in the form:

```
<expr>.<element-index>
```

Here, *<expr>* is any expression that has a tuple type, and *<element-index>* is either the *zero-based index* of the element or the name of an element in a named tuple.

Examples:

```
db> select (1, 'EdgeDB').0;
{1}

db> select (number := 1, name := 'EdgeDB').name;
```

(continues on next page)

(continued from previous page)

```
{"EdgeDB"}
```

```
db> select (number := 1, name := 'EdgeDB').1;
{"EdgeDB"}
```

### 5.13.3 Nesting tuples

Tuples can be nested:

```
db> select (nested_tuple := (1, 2)).nested_tuple.0;
{1}
```

Referencing a non-existent tuple element will result in an error:

```
db> select (1, 2).5;
EdgeQLError: 5 is not a member of a tuple
----- query context -----
line 1
    > select (1, 2).3;
```

### 5.13.4 Type syntax

A tuple type can be explicitly declared in an expression or schema declaration using the following syntax:

```
tuple "<" <element-type>, [<element-type>, ...] ">"
```

A named tuple:

```
tuple "<" <element-name> : <element-type> [, ...] ">"
```

Any type can be used as a tuple element type.

Here's an example of using this syntax in a schema definition:

```
type GameElement {
    required property name -> str;
    required property position -> tuple<x: int64, y: int64>;
}
```

```
type GameElement {
    required name: str;
    required position: tuple<x: int64, y: int64>;
}
```

Here's a few examples of using tuple types in EdgeQL queries:

```
db> select <tuple<int64, str>>('1', 3);
{(1, '3')}
db> select <tuple<x: int64, y: int64>>(1, 2);
```

(continues on next page)

(continued from previous page)

```
{(x := 1, y := 2)}
db> select (1, '3') is (tuple<int64, str>);
{true}
db> select ([1, 2], 'a') is (tuple<array<int64>, str>);
{true}
```

**type tuple****Index Keywords** tuple

A tuple type is a heterogeneous sequence of other types.

Tuple elements can optionally have names, in which case the tuple is called a *named tuple*.

Any type can be used as a tuple element type.

A tuple type is created implicitly when a *tuple constructor* is used:

```
db> select ('foo', 42);
{('foo', 42)}
```

Two tuples are equal if all of their elements are equal and in the same order. Note that element names in named tuples are not significant for comparison:

```
db> select (1, 2, 3) = (a := 1, b := 2, c := 3);
{true}
```

The syntax of a tuple type declaration can be found in [this section](#).

## 5.14 Ranges

**edb-alt-title** Range Functions and Operators

Ranges represent some interval of values. The intervals can include or exclude their boundaries or can even omit one or both boundaries. Only some scalar types have corresponding range types:

- range<int32>
- range<int64>
- range<float32>
- range<float64>
- range<decimal>
- range<datetime>
- range<cal::local\_datetime>
- range<cal::local\_date>

### 5.14.1 Constructing ranges

There's a special `range()` constructor function for making range values. This is a little different from how scalars, arrays and tuples are created typically in EdgeDB.

For example:

```
db> select range(1, 10);
{range(1, 10, inc_lower := true, inc_upper := false)}
db> select range(2.2, 3.3);
{range(2.2, 3.3, inc_lower := true, inc_upper := false)}
```

Broadly there are two kinds of ranges: *discrete* and *contiguous*. The discrete ranges are `range<int32>`, `range<int64>`, and `range<cal::local_date>`. All ranges over discrete types get normalized such that the lower bound is included (if present) and the upper bound is excluded:

```
db> select range(1, 10) = range(1, 9, inc_upper := true);
{true}
db> select range(1, 10) = range(0, 10, inc_lower := false);
{true}
```

Ranges over contiguous types don't have the same normalization mechanism because the underlying types don't have granularity which could be used to easily include or exclude a boundary value.

Sometimes a range cannot contain any values, this is called an *empty* range. These kinds of ranges can arise from performing various operations on them, but they can also be constructed. There are basically two equivalent ways of constructing an *empty* range. It can be explicitly constructed by providing the same upper and lower bounds and specifying that at least one of them is not *inclusive* (which is the default for all range constructors):

```
db> select range(1, 1);
{range({}, empty := true)}
```

Alternatively, it's possible to specify `{}` as a boundary and also provide the `empty := true` named-only argument. If the empty set is provided as a literal, it also needs to have a type cast, to specify which type of the range is being constructed:

```
db> select range(<int64>{}, empty := true);
{range({}, empty := true)}
```

Since empty ranges contain no values, they are all considered to be equal to each other (as long as the types are compatible):

```
db> select range(1, 1) = range(<int64>{}, empty := true);
{true}
db> select range(1, 1) = range(42.0, 42.0);
{true}

db> select range(1, 1) = range(<cal::local_date>{}, empty := true);
error: InvalidTypeError: operator '=' cannot be applied to operands of
type 'range<std::int64>' and 'range<cal::local_date>'
  - query:1:8
    |
1 |   select range(1, 1) = range(<cal::local_date>{}, empty := true);
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Consider using an explicit type cast or a conversion function.
```

## 5.14.2 JSON representation

Much like `arrays` and `tuples`, the range types cannot be directly cast to a `str`, but instead can be cast into a `json` structure:

```
db> select <json>range(1, 10);
{"inc_lower": true, "inc_upper": false, "lower": 1, "upper": 10}
```

It's also possible to cast in the other direction - from `json` to a specific range type:

```
db> select <range<int64>>to_json('{
...   "lower": 1,
...   "inc_lower": true,
...   "upper": 10,
...   "inc_upper": false
... }');
{range(1, 10, inc_lower := true, inc_upper := false)}
```

Empty ranges have a shorthand `json` representation:

```
db> select <json>range(<int64>{}, empty := true);
{"empty": true}
```

When casting from `json` to an empty range, all other fields may be omitted, but if they are present, they must be consistent with an empty range:

```
db> select <range<int64>>to_json('{"empty": true}');
{range({}, empty := true)}

db> select <range<int64>>to_json('{
...   "lower": 1,
...   "inc_lower": true,
...   "upper": 1,
...   "inc_upper": false
... }');
{range({}, empty := true)}

db> select <range<int64>>to_json('{
...   "lower": 1,
...   "inc_lower": true,
...   "upper": 1,
...   "inc_upper": false,
...   "empty": true
... }');
{range({}, empty := true)}

db> select <range<int64>>to_json('{
...   "lower": 1,
...   "inc_lower": true,
...   "upper": 2,
...   "inc_upper": false,
...   "empty": true
... }');
edgedb error: InvalidValueError: conflicting arguments in range
```

(continues on next page)

(continued from previous page)

`constructor: "empty" is ``true`` while the specified bounds suggest otherwise`

**Note:** When casting from `json` to a range the `lower` and `upper` fields are optional, but the `inclusivity` fields `inc_lower` and `inc_upper` are *mandatory*. This is to address the fact that whether the range boundaries are included by default can vary based on system or context and being explicit avoids subtle errors. The only exception to this are empty ranges that can have just the "empty": `true` field.

### 5.14.3 Functions and operators

<code>range &lt; range</code>	One range is before the other.
<code>range &gt; range</code>	One range is after the other.
<code>range &lt;= range</code>	One range is before or same as the other.
<code>range &gt;= range</code>	One range is after or same as the other.
<code>range + range</code>	Range union.
<code>range - range</code>	Range subtraction.
<code>range * range</code>	Range intersection.
<code>range()</code>	Construct a range.
<code>range_get_lower()</code>	Return lower bound value.
<code>range_get_upper()</code>	Return upper bound value.
<code>range_is_inclusive_lower()</code>	Check whether lower bound is inclusive.
<code>range_is_inclusive_upper()</code>	Check whether upper bound is inclusive.
<code>range_is_empty()</code>	Check whether a range is empty.
<code>range_unpack()</code>	Return values from a range.
<code>contains()</code>	Check if an element or a range is within another range.
<code>overlaps()</code>	Check whether ranges overlap.

### 5.14.4 Reference

**operator range<anypoint> < range<anypoint> -> bool**

One range is before the other.

Returns `true` if the lower bound of the first range is smaller than the lower bound of the second range. The unspecified lower bound is considered to be smaller than any specified lower bound. If the lower bounds are equal then the upper bounds are compared. Unspecified upper bound is considered to be greater than any specified upper bound.

```
db> select range(1, 10) < range(2, 5);
{true}
db> select range(1, 10) < range(1, 15);
{true}
db> select range(1, 10) < range(1);
{true}
db> select range(1, 10) < range(<int64>{}, 10);
{false}
```

An empty range is considered to come before any non-empty range.

```
db> select range(1, 10) < range(10, 10);
{false}
db> select range(1, 10) < range(<int64>{}, empty := true);
{false}
```

This is also how the `order by` clauses compares ranges.

#### **operator range<anypoint> > range<anypoint> -> bool**

One range is after the other.

Returns `true` if the lower bound of the first range is greater than the lower bound of the second range. The unspecified lower bound is considered to be smaller than any specified lower bound. If the lower bounds are equal then the upper bounds are compared. Unspecified upper bound is considered to be greater than any specified upper bound.

```
db> select range(1, 10) > range(2, 5);
{false}
db> select range(1, 10) > range(1, 5);
{true}
db> select range(1, 10) > range(1);
{false}
db> select range(1, 10) > range(<int64>{}, 10);
{true}
```

An empty range is considered to come before any non-empty range.

```
db> select range(1, 10) > range(10, 10);
{true}
db> select range(1, 10) > range(<int64>{}, empty := true);
{true}
```

This is also how the `order by` clauses compares ranges.

#### **operator range<anypoint> <= range<anypoint> -> bool**

One range is before or same as the other.

Returns `true` if the ranges are identical or if the lower bound of the first range is smaller than the lower bound of the second range. The unspecified lower bound is considered to be smaller than any specified lower bound. If the lower bounds are equal then the upper bounds are compared. Unspecified upper bound is considered to be greater than any specified upper bound.

```
db> select range(1, 10) <= range(1, 10);
{true}
db> select range(1, 10) <= range(2, 5);
{true}
db> select range(1, 10) <= range(1, 15);
{true}
db> select range(1, 10) <= range(1);
{true}
db> select range(1, 10) <= range(<int64>{}, 10);
{false}
```

An empty range is considered to come before any non-empty range.

```
db> select range(1, 10) <= range(10, 10);
{false}
db> select range(1, 1) <= range(10, 10);
{true}
db> select range(1, 10) <= range(<int64>{}, empty := true);
{false}
```

This is also how the `order by` clauses compares ranges.

---

#### **operator range<anypoint> >= range<anypoint> -> bool**

One range is after or same as the other.

Returns `true` if the ranges are identical or if the lower bound of the first range is greater than the lower bound of the second range. The unspecified lower bound is considered to be smaller than any specified lower bound. If the lower bounds are equal then the upper bounds are compared. Unspecified upper bound is considered to be greater than any specified upper bound.

```
db> select range(1, 10) >= range(2, 5);
{false}
db> select range(1, 10) >= range(1, 10);
{true}
db> select range(1, 10) >= range(1, 5);
{true}
db> select range(1, 10) >= range(1);
{false}
db> select range(1, 10) >= range(<int64>{}, 10);
{true}
```

An empty range is considered to come before any non-empty range.

```
db> select range(1, 10) >= range(10, 10);
{true}
db> select range(1, 1) >= range(10, 10);
{true}
db> select range(1, 10) >= range(<int64>{}, empty := true);
{true}
```

This is also how the `order by` clauses compares ranges.

#### **operator range<anypoint> + range<anypoint> -> range<anypoint>**

**Index Keywords** plus add

Range union.

Find the union of two ranges as long as the result is a single range without any discontinuities inside.

```
db> select range(1, 10) + range(5, 15);
{range(1, 15, inc_lower := true, inc_upper := false)}
db> select range(1, 10) + range(5);
{range(1, {}, inc_lower := true, inc_upper := false)}
```

---

#### **operator range<anypoint> - range<anypoint> -> range<anypoint>**

**Index Keywords** minus subtract

Range subtraction.

Subtract one range from another. This is only valid if the resulting range does not have any discontinuities inside.

```
db> select range(1, 10) - range(5, 15);
{range(1, 5, inc_lower := true, inc_upper := false)}
db> select range(1, 10) - range(<int64>{}, 5);
{range(5, 10, inc_lower := true, inc_upper := false)}
db> select range(1, 10) - range(0, 15);
{range({}, empty := true)}
```

**operator range<anypoint> \* range<anypoint>** -> range<anypoint>

**Index Keywords** intersect intersection

Range intersection.

Find the intersection of two ranges.

```
db> select range(1, 10) * range(5, 15);
{range(5, 10, inc_lower := true, inc_upper := false)}
db> select range(1, 10) * range(-15, 15);
{range(1, 10, inc_lower := true, inc_upper := false)}
db> select range(1) * range(-15, 15);
{range(1, 15, inc_lower := true, inc_upper := false)}
db> select range(10) * range(<int64>{}, 1);
{range({}, empty := true)}
```

**function std::range** → range<std::any point>

Construct a range.

Either one of *lower* or *upper* bounds can be set to {} to indicate an unbounded interval.

By default the *lower* bound is included and the *upper* bound is excluded from the range, but this can be controlled explicitly via the *inc\_lower* and *inc\_upper* named-only arguments.

```
db> select range(1, 10);
{range(1, 10, inc_lower := true, inc_upper := false)}
db> select range(1.5, 7.5, inc_lower := false);
{range(1.5, 7.5, inc_lower := false, inc_upper := false)}
```

Finally, an empty range can be created by using the *empty* named-only flag. The first argument still needs to be passed as an {} so that the type of the range can be inferred from it.

```
db> select range(<int64>{}, empty := true);
{range({}, empty := true)}
```

**function std::range\_get\_lower** → optional anypoint

Return lower bound value.

Return the lower bound of the specified range.

```
db> select range_get_lower(range(1, 10));
{1}
db> select range_get_lower(range(1.5, 7.5));
{1.5}
```

---

**function std::range\_is\_inclusive\_lower** → std::bool

Check whether lower bound is inclusive.

Return `true` if the lower bound is inclusive and `false` otherwise. If there is no lower bound, then it is never considered inclusive.

```
db> select range_is_inclusive_lower(range(1, 10));
{true}
db> select range_is_inclusive_lower(
...     range(1.5, 7.5, inc_lower := false));
{false}
db> select range_is_inclusive_lower(range(<int64>{}, 10));
{false}
```

---

**function std::range\_get\_upper** → optional anypoint

Return upper bound value.

Return the upper bound of the specified range.

```
db> select range_get_upper(range(1, 10));
{10}
db> select range_get_upper(range(1.5, 7.5));
{7.5}
```

---

**function std::range\_is\_inclusive\_upper** → std::bool

Check whether upper bound is inclusive.

Return `true` if the upper bound is inclusive and `false` otherwise. If there is no upper bound, then it is never considered inclusive.

```
db> select range_is_inclusive_upper(range(1, 10));
{false}
db> select range_is_inclusive_upper(
...     range(1.5, 7.5, inc_upper := true));
{true}
db> select range_is_inclusive_upper(range());
{false}
```

---

**function std::range\_is\_empty** → bool

Check whether a range is empty.

Return `true` if the range contains no values and `false` otherwise.

```
db> select range_is_empty(range(1, 10));
{false}
db> select range_is_empty(range(1, 1));
{true}
db> select range_is_empty(range(<int64>{}, empty := true));
{true}
```

**function std::range\_unpack** → set of anydiscrete  
**function std::range\_unpack** → set of anypoint

Return values from a range.

For a range of discrete values this function when called without indicating a *step* value simply produces a set of all the values within the range, in order.

```
db> select range_unpack(range(1, 10));
{1, 2, 3, 4, 5, 6, 7, 8, 9}
db> select range_unpack(range(
...   <cal::local_date>'2022-07-01',
...   <cal::local_date>'2022-07-10'));
{
  <cal::local_date>'2022-07-01',
  <cal::local_date>'2022-07-02',
  <cal::local_date>'2022-07-03',
  <cal::local_date>'2022-07-04',
  <cal::local_date>'2022-07-05',
  <cal::local_date>'2022-07-06',
  <cal::local_date>'2022-07-07',
  <cal::local_date>'2022-07-08',
  <cal::local_date>'2022-07-09',
}
```

For any range type a *step* value can be specified. Then the values will be picked from the range, starting at the lower boundary (skipping the boundary value itself if it's not included in the range) and then producing the next value by adding the *step* to the previous one.

```
db> select range_unpack(range(1.5, 7.5), 0.7);
{1.5, 2.2, 2.9, 3.6, 4.3, 5, 5.7, 6.4}
db> select range_unpack(
...   range(
...     <cal::local_datetime>'2022-07-01T00:00:00',
...     <cal::local_datetime>'2022-12-01T00:00:00'
...   ),
...   <cal::relative_duration>'25 days 5 hours');
{
  <cal::local_datetime>'2022-07-01T00:00:00',
  <cal::local_datetime>'2022-07-26T05:00:00',
  <cal::local_datetime>'2022-08-20T10:00:00',
  <cal::local_datetime>'2022-09-14T15:00:00',
  <cal::local_datetime>'2022-10-09T20:00:00',
  <cal::local_datetime>'2022-11-04T01:00:00',
}
```

**function std::overlaps** → std::bool

Check whether ranges overlap.

Return `true` if the ranges have any elements in common and `false` otherwise.

```
db> select overlaps(range(1, 10), range(5));
{true}
db> select overlaps(range(1, 10), range(10));
{false}
```

## 5.15 Bytes

**edb-alt-title** Bytes Functions and Operators

<code>bytes</code>	Byte sequence
<code>bytes[i]</code>	Accesses a byte at a given index.
<code>bytes[from:to]</code>	Produces a bytes sub-sequence from an existing bytes value.
<code>bytes ++ bytes</code>	Concatenates two bytes values into one.
<code>= != ?= ?!= &lt; &gt; &lt;= &gt;=</code>	Comparison operators
<code>len()</code>	Returns the number of bytes.
<code>contains()</code>	Checks if the byte sequence contains a given subsequence.
<code>find()</code>	Finds the index of the first occurrence of a subsequence.
<code>bytes_get_bit()</code>	Returns the specified bit of the bytes value.

---

**type bytes**

A sequence of bytes representing raw data.

Bytes can be represented as a literal using this syntax: `b''`.

```
db> select b'Hello, world';
{b'Hello, world'}
db> select b'Hello,\x20world\x01';
{b'Hello, world\x01'}
```

There are also some *generic* functions that can operate on bytes:

```
db> select contains(b'qwerty', b'42');
{false}
```

Bytes are rendered as base64-encoded strings in JSON. When you cast a bytes value into JSON, that's what you'll get. In order to *cast* a `json` value into bytes, it must be a base64-encoded string.

```
db> select <json>b'Hello EdgeDB!';
>{"\SGVsbG8gRWFnZURCIQ==\"}
db> select <bytes>to_json("{\"SGVsbG8gRWFnZURCIQ==\"");
{b'Hello EdgeDB!'}
```

---

**operator bytes [ int64 ] -> bytes**

Accesses a byte at a given index.

Examples:

```
db> select b'binary \x01\x02\x03\x04 ftw!'[2];
{b'n'}
db> select b'binary \x01\x02\x03\x04 ftw!'[8];
{b'\x02'}
```

**operator bytes [ int64 : int64 ] -> bytes**

Produces a bytes sub-sequence from an existing bytes value.

Examples:

```
db> select b'\x01\x02\x03\x04 ftw![2:-1];
{b'\x03\x04 ftw'}
db> select b'some bytes'[2:-3];
{b'me by'}
```

**operator bytes ++ bytes -> bytes**

Concatenates two bytes values into one.

```
db> select b'\x01\x02' ++ b'\x03\x04';
{b'\x01\x02\x03\x04'}
```

**function std::bytes\_get\_bit → int64**

Returns the specified bit of the bytes value.

When looking for the *n*th bit, this function will enumerate bits from least to most significant in each byte.

```
db> for n in {0, 1, 2, 3, 4, 5, 6, 7,
...     8, 9, 10, 11, 12, 13, 14, 15}
... union bytes_get_bit(b'ab', n);
{1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0}
```

## 5.16 Sequences

<code>sequence</code>	Auto-incrementing sequence of <code>int64</code> .
<code>sequence_next()</code>	Increments the given sequence to its next value and returns that value.
<code>sequence_reset()</code>	Resets a sequence to initial state or a given value, returning the value.

**type sequence**

An auto-incrementing sequence of `int64`.

This type can be used to create auto-incrementing *properties*:

```
scalar type TicketNo extending sequence;

type Ticket {
    property number -> TicketNo {
        constraint exclusive;
    }
}
```

```
scalar type TicketNo extending sequence;

type Ticket {
    number: TicketNo {
        constraint exclusive;
    }
}
```

A sequence is bound to the scalar type, not to the property, so if multiple properties use the same sequence, they will share the same counter. For each distinct counter, a separate scalar type that is extending `sequence` should be used.

---

#### **function std::sequence\_next** → int64

Increments the given sequence to its next value and returns that value.

See the note on [specifying your sequence](#) for best practices on supplying the `seq` parameter.

Sequence advancement is done atomically; each concurrent session and transaction will receive a distinct sequence value.

```
db> select sequence_next(introspect MySequence);
{11}
```

---

#### **function std::sequence\_reset** → int64

#### **function std::sequence\_reset** → int64

Resets a sequence to initial state or a given value, returning the value.

See the note on [specifying your sequence](#) for best practices on supplying the `seq` parameter.

The single-parameter form resets the sequence to its initial state, where the next `sequence_next()` call will return the first value in sequence. The two-parameter form allows you to set the current value of the sequence. The next `sequence_next()` call will return the value after the one you passed to `sequence_reset()`.

```
db> select sequence_reset(introspect MySequence);
{1}
db> select sequence_next(introspect MySequence);
{1}
db> select sequence_reset(introspect MySequence, 22);
{22}
db> select sequence_next(introspect MySequence);
{23}
```

---

**Note:** To specify the sequence to be operated on by either `sequence_next()` or `sequence_reset()`, you must pass a `schema::ScalarType` object. If the sequence argument is known ahead of time and does not change, we recommend passing it by using the `introspect` operator:

```
select sequence_next(introspect MySequenceType);
# or
select sequence_next(introspect typeof MyObj.seq_prop);
```

This style of execution will ensure that the reference to a sequential type from a given expression is tracked properly to guarantee schema referential integrity.

It doesn't work in every use case, though. If in your use case, the sequence type must be determined at run time via a query argument, you will need to query it from the `schema::ScalarType` set directly:

```
with
SeqType := (
    select schema::ScalarType
    filter .name = <str>$seq_type_name
)
select
    sequence_next(SeqType);
```

#### Warning: Caution

To work efficiently in high concurrency without lock contention, a `sequence_next()` execution is never rolled back, even if the containing transaction is aborted. This may result in gaps in the generated sequence. Likewise, the result of a `sequence_reset()` call is not undone if the transaction is rolled back.

## 5.17 Base Objects

<code>BaseObject</code>	Root object type
<code>Object</code>	Root for user-defined object types

`std::BaseObject` is the root of the object type hierarchy and all object types in EdgeDB, including system types, extend it either directly or indirectly. User-defined object types extend from `std::Object` type, which is a subtype of `std::BaseObject`.

### type `BaseObject`

The root object type.

Definition:

```
abstract type std::BaseObject {
    # Universally unique object identifier
    required property id -> uuid {
        default := (select std::uuid_generate_v1mc());
        readonly := true;
```

(continues on next page)

(continued from previous page)

```
    constraint exclusive;
}

# Object type in the information schema.
required readonly link __type__ -> schema::ObjectType;
}
```

Subtypes may override the `id` property, but only with a valid UUID generation function. Currently, these are `uuid_generate_v1mc()` and `uuid_generate_v4()`.

---

**type Object**

The root object type for user-defined types.

Definition:

```
abstract type std::Object extending std::BaseObject;
```

## 5.18 Abstract Types

Abstract types are used to describe polymorphic functions, otherwise known as “generic functions,” which can be called on a broad range of types.

---

**type anytype**

**Index Keywords** any anytype

A generic type.

It is a placeholder used in cases where no specific type requirements are needed, such as defining polymorphic parameters in functions and operators.

---

**type anyscalar**

**Index Keywords** any anytype scalar

An abstract base scalar type.

All scalar types are derived from this type.

---

**type anyenum**

**Index Keywords** any anytype enum

An abstract base enumerated type.

All `enum` types are derived from this type.

---

**type anytuple**

**Index Keywords** any anytype anytuple

A generic tuple.

Similarly to `anytype`, this type is used to denote a generic tuple without detailing its component types. This is useful when defining polymorphic parameters in functions and operators.

### 5.18.1 Abstract Numeric Types

These abstract numeric types extend `anyscalar`.

#### `type anyint`

**Index Keywords** any anytype int

An abstract base scalar type for `int16`, `int32`, and `int64`.

---

#### `type anyfloat`

**Index Keywords** any anytype float

An abstract base scalar type for `float32` and `float64`.

---

#### `type anyreal`

**Index Keywords** any anytype

An abstract base scalar type for `anyint`, `anyfloat`, and `decimal`.

### 5.18.2 Abstract Range Types

There are some types that can be used to construct `ranges`. These scalar types are distinguished by the following abstract types:

#### `type anypoint`

**Index Keywords** any anypoint anyrange

Abstract base type for all valid ranges.

Abstract base scalar type for `int32`, `int64`, `float32`, `float64`, `decimal`, `datetime`, `cal::local_datetime`, and `cal::local_date`.

---

#### `type anydiscrete`

**Index Keywords** any anydiscrete anyrange discrete

An abstract base type for all valid *discrete* ranges.

This is an abstract base scalar type for `int32`, `int64`, and `cal::local_date`.

---

#### `type anycontiguous`

**Index Keywords** any anycontiguous anyrange

An abstract base type for all valid *contiguous* ranges.

This is an abstract base scalar type for `float32`, `float64`, `decimal`, `datetime`, and `cal::local_datetime`.

## 5.19 Constraints

<code>exclusive</code>	Enforce uniqueness among all instances of the containing type
<code>expression</code>	Custom constraint expression
<code>one_of</code>	A list of allowable values
<code>max_value</code>	Maximum value numerically/lexicographically
<code>max_ex_value</code>	Maximum value numerically/lexicographically (exclusive range)
<code>max_len_value</code>	Maximum length (strings only)
<code>min_value</code>	Minimum value numerically/lexicographically
<code>min_ex_value</code>	Minimum value numerically/lexicographically (exclusive range)
<code>min_len_value</code>	Minimum length (strings only)
<code>regexp</code>	Regex constraint (strings only)

### `constraint std::expression`

A constraint based on an arbitrary boolean expression.

The expression constraint may be used as in this example to create a custom scalar type:

```
scalar type starts_with_a extending str {
    constraint expression on (__subject__[0] = 'A');
}
```

Example of using an expression constraint based on a couple of object properties to restrict maximum magnitude for a vector:

```
type Vector {
    required property x -> float64;
    required property y -> float64;
    constraint expression on (
        __subject__.x^2 + __subject__.y^2 < 25
    );
}
```

```
type Vector {
    required x: float64;
    required y: float64;
    constraint expression on (
        __subject__.x^2 + __subject__.y^2 < 25
    );
}
```

### `constraint std::one_of`

Specifies a list of allowed values.

Example:

```
scalar type Status extending str {
    constraint one_of ('Open', 'Closed', 'Merged');
}
```

### `constraint std::max_value`

Specifies the maximum allowed value.

Example:

```
scalar type max_100 extending int64 {
    constraint max_value(100);
}
```

**constraint std::max\_ex\_value**

Specifies a non-inclusive upper bound for the value.

Example:

```
scalar type maxex_100 extending int64 {
    constraint max_ex_value(100);
}
```

In the example above, in contrast to the `max_value` constraint, a value of the `maxex_100` type cannot be 100 since the valid range of `max_ex_value` does not include the value specified in the constraint.

**constraint std::max\_len\_value**

Specifies the maximum allowed length of a value.

Example:

```
scalar type Username extending str {
    constraint max_len_value(30);
}
```

**constraint std::min\_value**

Specifies the minimum allowed value.

Example:

```
scalar type non_negative extending int64 {
    constraint min_value(0);
}
```

**constraint std::min\_ex\_value**

Specifies a non-inclusive lower bound for the value.

Example:

```
scalar type positive_float extending float64 {
    constraint min_ex_value(0);
}
```

In the example above, in contrast to the `min_value` constraint, a value of the `positive_float` type cannot be 0 since the valid range of `min_ex_value` does not include the value specified in the constraint.

**constraint std::min\_len\_value**

Specifies the minimum allowed length of a value.

Example:

```
scalar type four_decimal_places extending int64 {
    constraint min_len_value(4);
}
```

**constraint std::regexp**

**Index Keywords** regex regexp regular

Limits to string values matching a regular expression.

Example:

```
scalar type LettersOnly extending str {
    constraint regexp(r'[A-Za-z]*');
}
```

See our documentation on [regular expression patterns](#) for more information on those.

#### **constraint std::exclusive**

Specifies that the link or property value must be exclusive (unique).

When applied to a `multi` link or property, the exclusivity constraint guarantees that for every object, the set of values held by a link or property does not intersect with any other such set in any other object of this type.

This constraint is only valid for concrete links and properties. Scalar type definitions cannot include this constraint.

Example:

```
type User {
    # Make sure user names are unique.
    required property name -> str {
        constraint exclusive;
    }

    # Make sure none of the "owned" items belong
    # to any other user.
    multi link owns -> Item {
        constraint exclusive;
    }
}
```

```
type User {
    # Make sure user names are unique.
    required name: str {
        constraint exclusive;
    }

    # Make sure none of the "owned" items belong
    # to any other user.
    multi owns: Item {
        constraint exclusive;
    }
}
```

Sometimes it's necessary to create a type where each combination of properties is unique. This can be achieved by defining an `exclusive` constraint for the type, rather than on each property:

```
type UniqueCoordinates {
    required property x -> int64;
    required property y -> int64;

    # Each combination of x and y must be unique.
```

(continues on next page)

(continued from previous page)

```

constraint exclusive on ( (.x, .y) );
}

type UniqueCoordinates {
    required x: int64;
    required y: int64;

    # Each combination of x and y must be unique.
    constraint exclusive on ( (.x, .y) );
}

```

In principle, many possible expressions can appear in the on (<expr>) clause of the exclusive constraint with a few caveats:

- The expression can only contain references to the immediate properties or links of the type.
- No *backlinks* or long paths are allowed.
- Only **Immutable** functions are allowed in the constraint expression.

---

**Note:** This constraint also has an additional effect of creating an implicit *index* on the link or property. This means that in the above example there's no need to add explicit indexes for the name property.

---

See also
<i>Schema &gt; Constraints</i>
<i>SDL &gt; Constraints</i>
<i>DDL &gt; Constraints</i>
<i>Introspection &gt; Constraints</i>
Tutorial > Advanced EdgeQL > Constraints

## 5.20 System

### edb-alt-title System Functions

<code>sys::get_version()</code>	Return the server version as a tuple.
<code>sys::get_version_as_str()</code>	Return the server version as a string.
<code>sys::get_current_database()</code>	Return the name of the current database as a string.

---

**function** `sys::get_version` → tuple<major: int64, minor: int64, stage: sys::VersionStage, stage\_no: int64, local: array<str>>

Return the server version as a tuple.

The `major` and `minor` elements contain the major and the minor components of the version; `stage` is an enumeration value containing one of 'dev', 'alpha', 'beta', 'rc' or 'final'; `stage_no` is the stage sequence number (e.g. 2 in an alpha 2 release); and `local` contains an arbitrary array of local version identifiers.

```

db> select sys::get_version();
{ (major := 1, minor := 0, stage := <sys::VersionStage>'alpha',
  stage_no := 1, local := [])}

```

---

```
function sys::get_version_as_str → str
    Return the server version as a string.
```

```
db> select sys::get_version_as_str();
{ '1.0-alpha.1'}
```

---

```
function sys::get_transaction_isolation → sys::TransactionIsolation
    Return the isolation level of the current transaction.
```

Possible return values are given by *sys::TransactionIsolation*.

```
db> select sys::get_transaction_isolation();
{sys::TransactionIsolation.Serializable}
```

---

```
function sys::get_current_database → str
    Return the name of the current database as a string.
```

```
db> select sys::get_current_database();
{ 'my_database'}
```

---

```
type sys::TransactionIsolation
```

**Index Keywords** enum transaction isolation

*Enum* indicating the possible transaction isolation modes.

This enum only accepts a value of Serializable.

## 5.21 Config

The `cfg` module contains a set of types and scalars used for configuring EdgeDB.

Type	Description
<code>cfg::Config</code>	The base type for all configuration objects. The properties of this type define the set of configuration settings supported by EdgeDB.
<code>cfg::Auth</code>	An object type representing an authentication profile.
<code>cfg::AuthMethod</code>	An abstract object type representing a method of authentication
<code>cfg::Trust</code>	A subclass of <code>AuthMethod</code> indicating an “always trust” policy (no authentication).
<code>cfg::SCRAM</code>	A subclass of <code>AuthMethod</code> indicating password-based authentication.
<code>cfg::memory</code>	A scalar type for storing a quantity of memory storage.

## 5.21.1 Configuration Parameters

**edb-alt-title** Available Configuration Parameters

### 5.21.1.1 Connection settings

**listen\_addresses** → **multi str** Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications. If the list is empty, the server does not listen on any IP interface at all.

**listen\_port** → **int16** The TCP port the server listens on; 5656 by default. Note that the same port number is used for all IP addresses the server listens on.

### 5.21.1.2 Resource usage

**effective\_ioConcurrency** → **int64** Sets the number of concurrent disk I/O operations that can be executed simultaneously. Corresponds to the PostgreSQL configuration parameter of the same name.

**query\_work\_mem** → **cfg::memory** The amount of memory used by internal query operations such as sorting. Corresponds to the PostgreSQL `work_mem` configuration parameter.

**shared\_buffers** → **cfg::memory** The amount of memory the database uses for shared memory buffers. Corresponds to the PostgreSQL configuration parameter of the same name. Changing this value requires server restart.

### 5.21.1.3 Query planning

**default\_statistics\_target** → **int64** Sets the default data statistics target for the planner. Corresponds to the PostgreSQL configuration parameter of the same name.

**effective\_cache\_size** → **cfg::memory** Sets the planner's assumption about the effective size of the disk cache that is available to a single query. Corresponds to the PostgreSQL configuration parameter of the same name.

### 5.21.1.4 Query behavior

**allow\_bare\_ddl** → **cfg::AllowBareDDL** Allows for running bare DDL outside a migration. Possible values are `cfg::AllowBareDDL.AlwaysAllow` and `cfg::AllowBareDDL.NeverAllow`.

When you create an instance, this is set to `cfg::AllowBareDDL.AlwaysAllow` until you run a migration. At that point it is set to `cfg::AllowBareDDL.NeverAllow` because it's generally a bad idea to mix migrations with bare DDL.

**apply\_access\_policies** → **bool** Determines whether access policies should be applied when running queries. Setting this to `false` effectively puts you into super-user mode, ignoring any access policies that might otherwise limit you on the instance.

---

**Note:** This setting can also be conveniently accessed via the “Config” dropdown menu at the top of the EdgeDB UI (accessible by running the CLI command `edgedb ui` from within a project). The setting will apply only to your UI session, so you won't have to remember to re-enable it when you're done.

---

### 5.21.1.5 Client connections

**allow\_user\_specified\_id -> bool** Makes it possible to set the `.id` property when inserting new objects.

Enabling this feature introduces some security vulnerabilities:

1. An unprivileged user can discover ids that already exist in the database by trying to insert new values and noting when there is a constraint violation on `.id` even if the user doesn't have access to the relevant table.
2. It allows re-using object ids for a different object type, which the application might not expect.

**session\_idle\_timeout -> std::duration** Sets the timeout for how long client connections can stay inactive before being forcefully closed by the server.

Time spent on waiting for query results doesn't count as idling. E.g. if the session idle timeout is set to 1 minute it would be OK to run a query that takes 2 minutes to compute; to limit the query execution time use the `query_execution_timeout` setting.

The default is 60 seconds. Setting it to `<duration>'0'` disables the mechanism. Setting the timeout to less than 2 seconds is not recommended.

Note that the actual time an idle connection can live can be up to two times longer than the specified timeout.

This is a system-level config setting.

**session\_idle\_transaction\_timeout -> std::duration** Sets the timeout for how long client connections can stay inactive while in a transaction.

The default is 10 seconds. Setting it to `<duration>'0'` disables the mechanism.

**query\_execution\_timeout -> std::duration** Sets a time limit on how long a query can be run.

Setting it to `<duration>'0'` disables the mechanism. The timeout isn't enabled by default.

---

### type cfg::Config

An abstract type representing the configuration of an instance or database.

The properties of this object type represent the set of configuration options supported by EdgeDB (listed above).

---

### type cfg::Auth

An object type designed to specify a client authentication profile.

```
edgedb> configure instance insert
.....
Auth {priority := 0, method := (insert Trust)};
OK: CONFIGURE INSTANCE
```

Below are the properties of the `Auth` class.

**priority -> int64** The priority of the authentication rule. The lower this number, the higher the priority.

**user -> multi str** The name(s) of the database role(s) this rule applies to. If set to `'*'`, then it applies to all roles.

**method -> cfg::AuthMethod** The name of the authentication method type. Expects an instance of `cfg::AuthMethod`; Valid values are: `Trust` for no authentication and `SCRAM` for SCRAM-SHA-256 password authentication.

**comment -> optional str** An optional comment for the authentication rule.

---

**type cfg::AuthMethod**

An abstract object class that represents an authentication method.

It currently has two concrete subclasses, each of which represent an available authentication method: `cfg::Trust` and `cfg::SCRAM`.

**type cfg::Trust**

The `cfg::Trust` indicates an “always-trust” policy.

When active, it disables password-based authentication.

```
edgedb> configure instance insert
..... Auth {priority := 0, method := (insert Trust)};
OK: CONFIGURE INSTANCE
```

**type cfg::SCRAM**

The `cfg::SCRAM` indicates password-based authentication.

This policy is implemented via SCRAM-SHA-256.

```
edgedb> configure instance insert
..... Auth {priority := 0, method := (insert SCRAM)};
OK: CONFIGURE INSTANCE
```

**type cfg::memory**

A scalar type representing a quantity of memory storage.

As with `uuid`, `datetime`, and several other types, `cfg::memory` values are declared by casting from an appropriately formatted string.

```
db> select <cfg::memory>'1B'; # 1 byte
{<cfg::memory>'1B'}
db> select <cfg::memory>'5KiB'; # 5 kibibytes
{<cfg::memory>'5KiB'}
db> select <cfg::memory>'128MiB'; # 128 mebibytes
{<cfg::memory>'128MiB'}
```

The numerical component of the value must be a non-negative integer; the units must be one of `B`|`KiB`|`MiB`|`GiB`|`TiB`|`PiB`. We’re using the explicit KiB unit notation (1024 bytes) instead of kB (which is ambiguous, and may mean 1000 or 1024 bytes).

## 5.22 Deprecated

### `edb-alt-title` Deprecated Functions

<code>str_lpad()</code>	Return the input string left-padded to the length n.
<code>str_rpad()</code>	Return the input string right-padded to the length n.
<code>str_ltrim()</code>	Return the input string with all leftmost trim characters removed.
<code>str_rtrim()</code>	Return the input string with all rightmost trim characters removed.

**function std::str\_lpad** → str

Return the input *string* left-padded to the length *n*.

**Warning:** This function is deprecated. Use `std::str_pad_start()` instead.

If the *string* is longer than *n*, then it is truncated to the first *n* characters. Otherwise, the *string* is padded on the left up to the total length *n* using *fill* characters (space by default).

```
db> select str_lpad('short', 10);
{'short'}
db> select str_lpad('much too long', 10);
{'much too l'}
db> select str_lpad('short', 10, '.:');
{'.:.:.short'}
```

---

**function std::str\_rpad** → str

Return the input *string* right-padded to the length *n*.

**Warning:** This function is deprecated. Use `std::str_pad_end()` instead.

If the *string* is longer than *n*, then it is truncated to the first *n* characters. Otherwise, the *string* is padded on the right up to the total length *n* using *fill* characters (space by default).

```
db> select str_rpad('short', 10);
{'short'}
db> select str_rpad('much too long', 10);
{'much too l'}
db> select str_rpad('short', 10, '.:');
{'short.....'}
```

---

**function std::str\_ltrim** → str

Return the input string with all leftmost *trim* characters removed.

**Warning:** This function is deprecated. Use `std::str_trim_start()` instead.

If the *trim* specifies more than one character they will be removed from the beginning of the *string* regardless of the order in which they appear.

```
db> select str_ltrim('    data');
{'data'}
db> select str_ltrim('.....data', '.:');
{'data'}
db> select str_ltrim(':::::data', '.:');
{'data'}
db> select str_ltrim('....:data', '.:');
{'data'}
db> select str_ltrim('....:data', '.:');
{'data'}
```

**function std::str\_rtrim** → str

Return the input string with all rightmost *trim* characters removed.

**Warning:** This function is deprecated. Use `std::str_trim_end()` instead.

If the *trim* specifies more than one character they will be removed from the end of the *string* regardless of the order in which they appear.

```
db> select str_rtrim('data      ');
{'data'}
db> select str_rtrim('data.....', '.:');
{'data'}
db> select str_rtrim('data:::::', '..');
{'data'}
db> select str_rtrim('data:::::', '..');
{'data'}
db> select str_rtrim('data....', '..');
{'data'}
```

EdgeDB comes with a rigorously defined type system consisting of **scalar types**, **collection types** (like arrays and tuples), and **object types**. There is also a library of built-in functions and operators for working with each datatype.

## 5.23 Scalar Types

*Scalar types* store primitive data.

- *Strings*
- *Numbers*
- *Booleans*
- *Dates and times*
- *Enums*
- *JSON*
- *UUID*
- *Bytes*
- *Sequences*
- *Abstract types*: these are the types that undergird the scalar hierarchy.

## 5.24 Collection Types

*Collection types* are special generic types used to group homogeneous or heterogeneous data.

- *Arrays*
- *Tuples*

## 5.25 Range Types

- *Range*

## 5.26 Object Types

- *Object Types*

## 5.27 Types and Sets

- *Sets*
- *Types*
- *Casting*

## 5.28 Utilities

- *Math*
- *Comparison*
- *Constraints*
- *System*

## CLIENT LIBRARIES

### Official Client Libraries

- Python
- TypeScript/JavaScript
- Go
- Rust
- .NET

### Community-Maintained Clients

- Elixir

### HTTP Protocols

- *EdgeQL over HTTP*
- *GraphQL over HTTP*

## 6.1 Connection

There are several ways to provide connection information to a client library.

- Use **projects**. This is the recommended approach for *local development*. Once the project is initialized, all client libraries that are running inside the project directory can auto-discover the project-linked instance, no need for environment variables or hard-coded credentials. Follow the [Using projects](#) guide to get started.
- Set the `EDGEDB_DSN` environment variable to a valid DSN (connection string). This is the recommended approach in *production*. A DSN is a connection URL of the form `edgedb://user:pass@host:port/database`. For a guide to DSNs, see the [DSN Specification](#).
- Set the `EDGEDB_INSTANCE` environment variable to a *name* of a local or remote linked instance. You can create new instances manually with the [edgedb instance create](#) command.
- Explicitly pass a DSN or *instance name* into the client creation function: `edgedb.createClient` in JS, `edgedb.create_client()` in Python, and `edgedb.CreateClient` in Go.

```
const client = edgedb.createClient({
  dsn: "edgedb://..."
});
```

Only use this approach in development; it isn't recommended to include sensitive information hard-coded in your production source code. Use environment variables instead. Different languages, frameworks, cloud hosting providers, and container-based workflows each provide various mechanisms for setting environment variables.

These are the most common ways to connect to an instance, however EdgeDB supports several other options for advanced use cases. For a complete reference on connection configuration, see [Reference > Connection Parameters](#).

## 6.2 JavaScript

The documentation for the JavaScript client is automatically pulled from <https://github.com/edgedb/edgedb-js/tree/master/docs> by the build pipeline of the edgedb.com website.

## 6.3 Python

The documentation for the Python client is automatically pulled from <https://github.com/edgedb/edgedb-python/tree/master/docs> by the build pipeline of the edgedb.com website.

## 6.4 Go

The documentation for the Go client is automatically generated from <https://github.com/edgedb/edgedb-go> by the build pipeline of the edgedb.com website.

## 6.5 Rust

### edb-alt-title EdgeDB Rust Client

EdgeDB maintains an client library for Rust. View the [full documentation](#).

```
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let conn = edgedb_tokio::create_client().await?;
    let val = conn.query_required_single::<i64, _>(
        "SELECT 7*8",
        &(),
    ).await?;
    println!("7*8 is: {}", val);
    Ok(())
}
```

## 6.6 Dart

The documentation for the Dart client is automatically generated from <https://github.com/edgedb/edgedb-dart> by the build pipeline of the edgedb.com website.

## 6.7 .NET

The documentation for the .NET client is automatically generated from <https://github.com/edgedb/edgedb-net/tree/dev/docs> by the build pipeline of the edgedb.com website.

## 6.8 EdgeQL over HTTP

EdgeDB can expose an HTTP endpoint for EdgeQL queries. Since HTTP is a stateless protocol, no *DDL*, *transaction commands*, can be executed using this endpoint. Only one query per request can be executed.

In order to set up HTTP access to the database add the following to the schema:

```
using extension edgeql_http;
```

Then create a new migration and apply it using *edgedb migration create* and *edgedb migrate*, respectively.

Your instance can now receive EdgeQL queries over HTTP at `http://<hostname>:<port>/db/<database-name>/edgeql`.

**Note:** Here's how to determine your local EdgeDB instance's HTTP server URL:

- The hostname will be `localhost`
- Find the port by running `edgedb instance list`. This will print a table of all EdgeDB instances on your machine, including their associated port number.
- In most cases, `database_name` will be `edgedb`. An EdgeDB *instance* can contain multiple databases. On initialization, a default database called `edgedb` is created; all queries are executed against this database unless otherwise specified.

To determine the URL of a remote instance you have linked with the CLI, you can get both the hostname and port of the instance from the “Port” column of the `edgedb instance list` table (formatted as `<hostname>:<port>`). The same guidance on local database names applies here.

### 6.8.1 Protocol

EdgeDB supports GET and POST methods for handling EdgeQL over HTTP protocol. Both GET and POST methods use the following fields:

- `query` - contains the EdgeQL query string
- `variables` - contains a JSON object where the keys are the parameter names from the query and the values are the arguments to be used in this execution of the query.
- `globals` - contains a JSON object where the keys are the fully qualified global names and the values are the desired values for those globals.

The protocol supports HTTP Keep-Alive.

### 6.8.1.1 GET request

The HTTP GET request passes the fields as query parameters: `query` string and JSON-encoded `variables` mapping.

### 6.8.1.2 POST request

The POST request should use `application/json` content type and submit the following JSON-encoded form with the necessary fields:

```
{  
    "query": "...",  
    "variables": { "varName": "varValue", ... },  
    "globals": {"default::global_name": "value"}  
}
```

### 6.8.1.3 Response

The response format is the same for both methods. The body of the response is JSON of the following form:

```
{  
    "data": [ ... ],  
    "error": {  
        "message": "Error message",  
        "type": "ErrorType",  
        "code": 123456  
    }  
}
```

The `data` response field will contain the response set serialized as a JSON array.

Note that the `error` field will only be present if an error actually occurred. The `error` will further contain the `message` field with the error message string, the `type` field with the name of the type of error and the `code` field with an integer `error code`.

---

**Note:** Caution is advised when reading `decimal` or `bigint` values using HTTP protocol because the results are provided in JSON format. The JSON specification does not have a limit on significant digits, so a `decimal` or a `bigint` number can be losslessly represented in JSON. However, JSON decoders in many languages will read all such numbers as some kind of 32- or 64-bit number type, which may result in errors or precision loss. If such loss is unacceptable, then consider casting the value into `str` and decoding it on the client side into a more appropriate type.

---

## 6.8.2 Health Checks

EdgeDB exposes HTTP endpoints to check for aliveness and readiness of your database instance. You can make GET requests to these endpoints to check the instance status.

### 6.8.2.1 Aliveness

Check that your instance is alive by making a request to `http://<hostname>:<port>/server/status/alive`. If your instance is alive, it will respond with a `200` status code and "OK" as the payload. Otherwise, it will respond with a `50x` or a network error.

### 6.8.2.2 Readiness

Check that your instance is ready by making a request to `http://<hostname>:<port>/server/status/ready`. If your instance is ready, it will respond with a `200` status code and "OK" as the payload. Otherwise, it will respond with a `50x` or a network error.

## 6.9 GraphQL

### 6.9.1 Basics

For the purposes of this section, we will consider a `default` module containing the following schema:

```
type Author {
    property name -> str;
}

type Book {
    # to make the examples simpler only the title is
    # a required property
    required property title -> str;
    property synopsis -> str;
    link author -> Author;
    property isbn -> str {
        constraint max_len_value(10);
    }
}
```

From the schema above, EdgeDB will expose to GraphQL:

- object types `Author` and `Book`
- scalars `String` and `ID`

In addition to this, the `Query` will have two fields — `Author`, and `Book` — to query these types respectively.

#### 6.9.1.1 Queries

Consider this example:

GraphQL	EdgeQL equivalent
<pre>{   Book {     title     synopsis     author {       name     }   } }</pre>	<pre>select   Book {     title,     synopsis,     author: {       name     } };</pre>

The top-level field of the GraphQL query must be a valid name of an object type or an expression alias of something returning an object type. Nested fields must be valid links or properties.

There are some specific conventions as to how *arguments* in GraphQL queries are used to allow filtering, ordering, and paginating data.

#### 6.9.1.1.1 Filtering

Filtering the retrieved data is done by specifying a `filter` argument. The `filter` argument is customized to each specific type based on the available fields. In case of the sample schema, here are the specifications for available filter arguments for querying Book:

```
# this is Book-specific
input FilterBook {
    # basic boolean operators that combine conditions
    and: [FilterBook!]
    or: [FilterBook!]
    not: FilterBook

    # fields available for filtering (properties in EdgeQL)
    title: FilterString
    synopsis: FilterString
    isbn: FilterString
    author: NestedFilterAuthor
}

# this is Author-specific
input NestedFilterAuthor {
    # instead of boolean operations, "exists" check is available
    # for links
    exists: Boolean

    # fields available for filtering (properties in EdgeQL)
    name: FilterString
}

# this is generic
input FilterString {
    # "exists" check is available for every property, too
```

(continues on next page)

(continued from previous page)

```

exists: Boolean

# equality
eq: String
neq: String

# lexicographical comparison
gt: String
gte: String
lt: String
lte: String

# other useful operations
like: String
ilike: String
}

```

Here are some examples of using a filter:

GraphQL	EdgeQL equivalent
<pre> {   Book(     filter: {       title: {         eq: "Spam"       }     }   ) {     title     synopsis   } } </pre>	<pre> select   Book {     title,     synopsis   } filter   Book.title = 'Spam'; </pre>
<pre> {   Book(     filter: {       author: {         name: {           eq:             "Lewis Carroll"         }       }     }   ) {     title     synopsis   } } </pre>	<pre> select   Book {     title,     synopsis   } filter   Book.author.name =     'Lewis Carroll'; </pre>

It is legal to provide multiple input fields in the same input object. They are all implicitly combined using a logical

conjunction. For example:

GraphQL	EdgeQL equivalent
<pre>{   Book(     filter: {       title: {         gte: "m",         lt: "o"       }     }   ) {     title   } }</pre>	<pre>select   Book {     title,   } filter   Book.title &gt;= 'm'   and   Book.title &lt; 'o';</pre>

It is possible to search for books that don't specify the author:

GraphQL	EdgeQL equivalent
<pre>{   Book(     filter: {       author: {         exists: false       }     }   ) {     id     title   } }</pre>	<pre>select   Book {     id,     title   } filter   not exists     Book.author;</pre>

### 6.9.1.1.2 Ordering

Ordering the retrieved data is done by specifying an `order` argument. The `order` argument is customized to each specific type based on the available fields, much like the `filter`. In case of the sample schema, here are the specifications for the available filter arguments:

```
# this is Author-specific
input OrderAuthor {
  # fields available for ordering (properties in EdgeQL)
  name: Ordering
}

# this is Book-specific
input OrderBook {
  # fields available for ordering (properties in EdgeQL)
  title: Ordering
```

(continues on next page)

(continued from previous page)

```

synopsis: Ordering
isbn: Ordering
}

# this is generic
input Ordering {
    dir: directionEnum
    nulls: nullsOrderingEnum
}

enum directionEnum {
    ASC
    DESC
}

enum nullsOrderingEnum {
    SMALLEST    # null < any other value
    BIGGEST     # null > any other value
}

```

If the value of `nulls` is not specified it is assumed to be `SMALLEST`.

GraphQL	EdgeQL equivalent
<pre> {     Author(         order: {             name: {                 dir: ASC,                 nulls: BIGGEST             }         }     ) {         name     } } </pre>	<pre> select     Author {         name,     } order by     Author.name asc     empty last; </pre>

### 6.9.1.1.3 Paginating

Paginating the retrieved data is done by providing one or more of the following arguments: `first`, `last`, `before`, and `after`. The pagination works in a similar way to Relay Connections. In case of the sample schema, here are the specifications for the available filter arguments:

```

# a relevant Query definition snippet
type Query {
    Author(
        filter: FilterAuthor,
        order: OrderAuthor,
        after: String,

```

(continues on next page)

(continued from previous page)

```
    before: String,  
    first: Int,  
    last: Int,  
): [Author!]  
  
    # ... other Query fields  
}
```

The `after` and `before` strings are, in fact, string representations of numeric indices under the particular filter and ordering (starting at “0”). This makes the usage fairly intuitive even without having Relay Connection edges and cursors.

The objects corresponding to the indices specified by `before` or `after` are not included.

GraphQL	EdgeQL equivalent
<pre>{   Author(     order: {       name: {         dir: ASC       }     },     first: 10   ) {     name   } }</pre>	<pre>select   Author {     name,   } order by   Author.name asc limit 10;</pre>
<pre>{   Author(     order: {       name: {         dir: ASC       }     },     after: "19",     first: 10   ) {     name   } }</pre>	<pre>select   Author {     name,   } order by   Author.name asc offset 20 limit 10;</pre>
<pre>{   Author(     order: {       name: {         dir: ASC       }     },     after: "19",     before: "30"   ) {     name   } }</pre>	<pre>select   Author {     name,   } order by   Author.name asc offset 20 limit 10;</pre>

### 6.9.1.1.4 Variables

It is possible to use variables within GraphQL queries. They are mapped to variables in EdgeQL.

GraphQL	EdgeQL equivalent
<pre>query (\$title: String!) {     Book(         filter: {             title: {                 eq: \$title             }         }     ) {         title         synopsis     } }</pre>	<pre>select     Book {         title,         synopsis,     } filter     .title = \$title;</pre>

## 6.9.2 Mutations

EdgeDB provides GraphQL mutations to perform `delete`, `insert` and `update` operations.

### 6.9.2.1 Delete

The “delete” mutation is very similar in structure to a query. Basically, it works the same way as a query, using the `filter`, `order`, and various *pagination parameters* to define a set of objects to be deleted. These objects are also returned as the result of the delete mutation. Each object type has a corresponding `delete_<type>` mutation:

GraphQL	EdgeQL equivalent
<pre> <b>mutation</b> delete_all_books {     delete_Book {         title         synopsis         author {             name         }     } } </pre>	<pre> <b>select</b> (     <b>delete</b> Book ) {     title,     synopsis,     author: {         name     } }; </pre>
<pre> <b>mutation</b> delete_book_spam {     delete_Book(         filter: {             title: {                 eq: "Spam"             }         }     ) {         title         synopsis     } } </pre>	<pre> <b>select</b> (     <b>delete</b> Book     <b>filter</b>         Book.title = 'Spam' ) {     title,     synopsis }; </pre>
<pre> <b>mutation</b> delete_one_book {     delete_Book(         filter: {             author: {                 name: {                     eq:                         "Lewis Carroll"                 }             },             order: {                 title: {                     dir: ASC                 }             },             first: 1         )     ) {         title         synopsis     } } </pre>	<pre> <b>select</b> (     <b>delete</b> Book     <b>filter</b>         Book.author.name =             'Lewis Carroll'     <b>order by</b>         Book.title <b>ASC</b>     <b>limit</b> 1 ) {     title,     synopsis }; </pre>

### 6.9.2.2 Insert

The “insert” mutation exists for every object type. It allows creating new objects and supports nested insertions, too. The objects to be inserted are specified via the `data` parameter, which takes a list of specifications. Each such specification has the same structure as the object being inserted with required and optional fields (although if a field is required in the object but has a default, it’s optional in the insert specification):

GraphQL	EdgeQL equivalent
<pre><code>mutation insert_books {     insert_Book(         data: [{             title: "One"         }, {             title: "Two"         }]     ) {         id         title     } }</code></pre>	<pre><code>select {     insert Book {         title := "One"     },     insert Book {         title := "Two"     } } {     id,     title };</code></pre>

It’s possible to insert a nested structure all at once (e.g., a new book and a new author):

GraphQL	EdgeQL equivalent
<pre><code>mutation insert_books {     insert_Book(         data: [{             title: "Three",             author: {                 data: {                     name:                     "Unknown"                 }             }         }]     ) {         id         title     } }</code></pre>	<pre><code>select (     insert Book {         title := "Three",         author := (             insert Author {                 name :=                 "Unknown"             }         )     } ) {     id,     title };</code></pre>

It’s also possible to insert a new object that’s connected to an existing object (e.g. a new book by an existing author). In this case the nested object is specified using `filter`, `order`, and various `pagination parameters` to define a set of objects to be connected:

GraphQL	EdgeQL equivalent
<pre> <b>mutation</b> insert_book {   insert_Book(     data: [       {         title: "Four",         author: {           filter: {             name: {eq: "Unknown"}           }         }       }     ]   ) {     id     title   } } </pre>	<pre> <b>select</b> (   <b>insert</b> Book {     title := "Four",     author := (       <b>select</b> Author       <b>filter</b>         Author.name =         "Unknown"     )   ) ) {   id,   title }; </pre>

### 6.9.2.3 Update

The “update” mutation has features that are similar to both an “insert” mutation and a query. On one hand, the mutation takes *filter*, *order*, and various *pagination parameters* to define a set of objects to be updated. On the other hand, the *data* parameter is used to specify what and how should be updated.

The *data* parameter contains the fields that should be altered as well as what type of update operation must be performed (*set*, *increment*, *append*, etc.). The particular operations available depend on the type of field being updated.

GraphQL	EdgeQL equivalent
<pre> <b>mutation</b> update_book {     update_Book(         filter: {             title: {                 eq: "One"             }         }     data: {         synopsis: {             set: "TBD"         }         author: {             set: {                 filter: {                     name: {                         eq:                             "Unknown"                     }                 }             }         }     } ) {     id     title } } </pre>	<pre> <b>with</b>     Upd := (         <b>update</b> Book         <b>filter</b>             Book.title =                 "One"     <b>set</b> {         synopsis :=             "TBD",         author := (             <b>select</b> Author             <b>filter</b>                 Author.name =                     "Unknown"         )     ) <b>select</b> Upd {     id,     title }; </pre>

### 6.9.3 Introspection

GraphQL introspection can be used to explore the exposed EdgeDB types and expression aliases. Note that there are certain types like `tuple` that cannot be expressed in terms of the GraphQL type system (a `tuple` can be like a heterogeneous “List”).

Consider the following GraphQL introspection query:

```
{
  __type(name: "Query") {
    name
    fields {
      name
      args {
        name
        type {
          kind
          name
        }
      }
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}
```

Produces:

```
{
  "__type": {
    "name": "Query",
    "fields": [
      {
        "name": "Author",
        "args": [
          {
            "name": "id",
            "type": {
              "kind": "SCALAR",
              "name": "ID"
            }
          },
          {
            "name": "name",
            "type": {
              "kind": "SCALAR",
              "name": "String"
            }
          }
        ]
      },
      {
        "name": "Book",
        "args": [
          {
            "name": "id",
            "type": {
              "kind": "SCALAR",
              "name": "ID"
            }
          },
          {
            "name": "isbn",
            "type": {
              "kind": "SCALAR",
              "name": "String"
            }
          },
          {
            "name": "synopsis",
            "type": {
              "kind": "SCALAR",
              "name": "String"
            }
          }
        ],
      }
    ]
  }
}
```

(continues on next page)

(continued from previous page)

```
{
    "name": "title",
    "type": {
        "kind": "SCALAR",
        "name": "String"
    }
}
]
}
}
}
```

The above example shows what has been exposed for querying with GraphQL.

#### 6.9.4 Cheatsheet

---

**Note:** The types used as examples in these queries are defined *in our “Object types” cheatsheet*.

---

In order to set up GraphQL access to the database, add the following to the schema:

```
using extension graphql;
```

Then create a new migration and apply it using `edgedb migration create` and `edgedb migrate`, respectively.

Select all users in the system:

```
{
    User {
        id
        name
        image
    }
}
```

Select a movie by title and release year with associated actors ordered by last name:

```
{
    Movie(
        filter: {
            title: {eq: "Dune"},
            year: {eq: 2020}
        }
    ) {
        id
        title
        year
        description
    }
}
```

(continues on next page)

(continued from previous page)

```

directors {
    id
    full_name
}

actors(order: {last_name: {dir: ASC}}) {
    id
    full_name
}
}
}
```

Select movies with Keanu Reeves:

```
{
    Movie(
        filter: {
            actors: {full_name: {eq: "Keanu Reeves"}}
        }
    ) {
        id
        title
        year
        description
    }
}
```

Select a movie by title and year with top 3 most recent reviews (this uses *MovieAlias* in order to access reviews):

```
{
    MovieAlias(
        filter: {
            title: {eq: "Dune"},
            year: {eq: 2020}
        }
    ) {
        id
        title
        year
        description
        reviews(
            order: {creation_time: {dir: DESC}},
            first: 3
        ) {
            id
            body
            rating
            creation_time
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

        author {
            id
            name
        }
    }
}

```

Use *MovieAlias* in order to find movies that have no reviews:

```
{
  MovieAlias(
    filter: {
      reviews: {exists: false},
    }
  ) {
    id
    title
    year
    description
  }
}
```

Use a GraphQL *mutation* to add a user:

```
mutation add_user {
  insert_User(
    data: {name: "Atreides", image: "atreides.jpg"}
  ) {
    id
  }
}
```

Use a GraphQL *mutation* to add a review by an existing user:

```
mutation add_review {
  insert_Review(
    data: {
      # Since the movie already exists,
      # we select it using the same filter
      # mechanism as for queries.
      movie: {
        filter: {title: {eq: "Dune"}, year: {eq: 2020}},
        first: 1
      },
      body: "Yay!",
      rating: 5,
      # Similarly to the movie we select
    }
  )
}
```

(continues on next page)

(continued from previous page)

```

# the existing user.
author: {
    filter: {name: {eq: "Atreides"}},
    first: 1
}
)
{
    id
    body
}
}

```

Use a GraphQL *mutation* to add an actress to a movie:

```

mutation add_actor {
    update_Movie(
        # Specify which movie needs to be updated.
        filter: {title: {eq: "Dune"}, year: {eq: 2020}},
        # Specify the movie data to be updated.
        data: {
            actors: {
                add: [
                    filter: {
                        full_name: {eq: "Charlotte Rampling"}
                    }
                ]
            }
        )
    {
        id
        actors {
            id
        }
    }
}

```

EdgeDB supports GraphQL queries via the built-in `graphql` extension. A full CRUD API for all object types, their properties (both material and computed), their links, and all `aliases` is reflected in the GraphQL schema.

## 6.9.5 Setting up the extension

In order to set up GraphQL access to the database, add the following to the schema:

```
using extension graphql;
```

Then create a new migration and apply it.

```
$ edgedb migration create
$ edgedb migrate
```

Refer to the [connection docs](#) for various methods of running these commands against remotely-hosted instances.

## 6.9.6 Connection

Once you've activated the extension, your instance will listen for incoming GraphQL queries via HTTP at the following URL.

`http://127.0.0.1:<instance-port>/db/<database-name>/graphql`

The value of `<database-name>` is probably `edgedb`, which is the name of the default database that is created when an instance is first created. (If you've manually created additional databases, use the name of the database you'd like to query instead.)

To find the port number associated with a local instance, run `edgedb instance list`.

```
$ edgedb instance list
```

Kind	Name	Port	Version	Status	
local	inst1	10700	2.x	running	
local	inst2	10702	2.x	running	
local	inst3	10703	2.x	running	

To execute a GraphQL query against the database `edgedb` on the instance named `inst2`, we would send an HTTP request to `http://localhost:10702/db/edgedb/graphql`.

---

**Note:** The endpoint also provides a [GraphiQL](#) interface to explore the GraphQL schema and write queries. Take the GraphQL query endpoint, append `/explore`, and visit that URL in the browser. Under the above example, the GraphiQL endpoint is available at `http://localhost:10702/db/edgedb/graphql/explore`.

---

But what kind of HTTP request should this be? And what data should it contain?

## 6.9.7 The protocol

EdgeDB can receive GraphQL queries via both GET and POST requests. Requests can contain the following fields:

- `query` - the GraphQL query string
- `variables` - a JSON object containing a set of variables. **Optional** If the GraphQL query string contains variables, the `variables` object is required.
- `globals` - a JSON object containing global variables. **Optional**. The keys must be the fully qualified names of the globals to set (e.g., `default::current_user` for the global `current_user` in the `default` module).
- `operationName` - the name of the operation that must be executed. **Optional** If the GraphQL query contains several named operations, it is required.

---

**Note:** The protocol implementations conform to the official GraphQL [HTTP protocol](#). The protocol supports HTTP Keep-Alive.

---

### 6.9.7.1 POST request (recommended)

The POST request should use `application/json` content type and submit the following JSON-encoded form with the necessary fields.

```
$ curl \
  -H "Content-Type: application/json" \
  -X POST http://localhost:10787/db/edgedb/graphql \
  -d '{ "query": "query getMovie($title: String!) { Movie(filter: {title:{eq: $title}} \
  ↵) { id title }}", "variables": { "title": "The Batman" }, "globals": { \
  ↵"default::current_user": "04e52807-6835-4eaa-999b-952804ab40a5"}' \
  {"data": {...}}'
```

### 6.9.7.2 GET request

When using GET requests, any values for `query`, `variables`, `globals`, or `operationName` should be passed as query parameters in the URL.

```
$ curl \
  -H application/x-www-form-urlencoded \
  -X GET http://localhost:10787/db/edgedb/graphql \
  -G \
  --data-urlencode 'query=query getMovie($title: String!) { Movie(filter: {title:{eq: \
  ↵$title}} { id title }}' \
  --data-urlencode 'variables={ "title": "The Batman" }' \
  --data-urlencode 'globals={ "default::current_user": "04e52807-6835-4eaa-999b- \
  ↵952804ab40a5" }' \
  {"data": {...}}'
```

### 6.9.7.3 Response format

The body of the response is JSON in the following format:

```
{
  "data": { ... },
  "errors": [
    { "message": "Error message"}, ...
  ]
}
```

Note that the `errors` field will only be present if some errors actually occurred.

---

**Note:** Caution is advised when reading `decimal` or `bigint` values (mapped onto `Decimal` and `Bigint` GraphQL custom scalar types) using HTTP protocol because the results are provided in JSON format. The JSON specification does not have a limit on significant digits, so a `decimal` or a `bigint` number can be losslessly represented in JSON. However, JSON decoders in many languages will read all such numbers as some kind of 32- or 64-bit number type, which may result in errors or precision loss. If such loss is unacceptable, then consider creating a computed property which casts the value into `str` and decoding it on the client side into a more appropriate type.

---

## 6.9.8 Known limitations

We provide this GraphQL extension to support users who are accustomed to writing queries in GraphQL. That said, GraphQL is quite limited and verbose relative to EdgeQL.

There are also some additional limitations:

- Variables can only correspond to *scalar types*; you can't use GraphQL *input* types. Under the hood, query variables are mapped onto EdgeQL parameters, which only support scalar types.

As a consequence of this, you must declare top-level variables for each property for a GraphQL insertion mutation, which can make queries more verbose.

- Due to the differences between EdgeQL and GraphQL syntax, *enum* types which have values that cannot be represented as GraphQL identifiers (e.g. `N/A` or `NOT\_APPLICABLE`) cannot be properly reflected into GraphQL enums.
- Inserting or updating tuple properties is not yet supported.
- *Link properties* are not reflected, as GraphQL has no such concept.
- Every non-abstract EdgeDB object type is simultaneously an interface and an object in terms of the GraphQL type system, which means that, for every one object type name, two names are needed in reflected GraphQL. This potentially results in name clashes if the convention of using camel-case names for user types is not followed in EdgeDB.

## CLI

### **edb-alt-title** The EdgeDB CLI

The `edgedb` command-line interface (CLI) provides an idiomatic way to install EdgeDB, spin up local instances, open a REPL, execute queries, manage auth roles, introspect schema, create migrations, and more.

You can install it with one shell command.

### Installation

On Linux or MacOS, run the following in your terminal and follow the on-screen instructions:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.edgedb.com | sh
```

For Windows, the installation script is:

```
PS> iwr https://ps1.edgedb.com -useb | iex
```

- The `script`, inspired by `rustup`, will detect the OS and download the appropriate build of the EdgeDB CLI tool, `edgedb`.
- The `edgedb` command is a single executable (it's open source!)
- Once installed, the `edgedb` command can be used to install, uninstall, upgrade, and interact with EdgeDB server instances.
- You can uninstall EdgeDB server or remove the `edgedb` command at any time.

### Connection options

All commands respect a common set of *connection options*, which let you specify a target instance. This instance can be local to your machine or hosted remotely.

### Nightly version

To install the nightly version of the CLI (not to be confused with the nightly version of EdgeDB itself!) use this command:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.edgedb.com | \
sh -s -- --nightly
```

## Uninstallation

Command-line tools contain just one binary, so to remove it on Linux or macOS run:

```
$ rm "$(which edgedb)"
```

To remove all configuration files, run `edgedb info` to list the directories where EdgeDB stores data, then use `rf -rf <dir>` to delete those directories.

If the command-line tool was installed by the user (recommended) then it will also remove the binary.

If you've used `edgedb` commands you can also delete `instances` and `server` packages, prior to removing the tool:

```
$ edgedb instance destroy <instance_name>
```

To list instances and server versions use the following commands respectively:

```
$ edgedb instance status
$ edgedb server list-versions --installed-only
```

## Configure CLI and REPL

You can customize the behavior of the `edgedb` CLI and REPL with a global configuration file. The file is called `cli.toml` and its location differs between operating systems. Use `edgedb info` to find the “Config” directory on your system.

The `cli.toml` has the following structure. All fields are optional:

```
[shell]
expand-strings = true          # Stop escaping newlines in quoted strings
history-size = 10000            # Set number of entries retained in history
implicit-properties = false    # Print implicit properties of objects
limit = 100                     # Set implicit LIMIT
                                 # Defaults to 100, specify 0 to disable
input-mode = "emacs"            # Set input mode. One of: vi, emacs
output-format = "default"       # Set output format.
                                 # One of: default, json, json-pretty,
                                 # json-lines
print-stats = "off"             # Print statistics on each query.
                                 # One of: off, query, detailed
verbose-errors = false          # Print all errors with maximum verbosity
```

*Notes on network usage*

## 7.1 Connection flags

The `edgedb` CLI supports a standard set of connection flags used to specify the *target* of a given command. The CLI always respects any connection parameters passed explicitly using flags.

- If no flags are provided, then environment variables will be used to determine the instance.
- If no environment variables are present, the CLI will check if the working directory is within an instance-linked project directory.
- If none of the above are present, the command fails.

For a detailed breakdown of how connection information is resolved, read the [Connection Parameter Resolution](#) docs.

### 7.1.1 Connection flags

**-I <name>, --instance=<name>** Specifies the named instance to connect to. The actual connection parameters are stored in <edgedb\_config\_dir>/credentials and are usually created by `edgedb instance create` or similar commands. Run `edgedb info` to see the location of <edgedb\_config\_dir> on your machine.

This option overrides host and port.

**--dsn=<dsn>** Specifies the DSN for EdgeDB to connect to.

This option overrides all other options except password.

**--credentials-file /path/to/file** Path to JSON file containing credentials.

**-H <hostname>, --host=<hostname>** Specifies the host name of the machine on which the server is running. Defaults to the value of the EDGEDB\_HOST environment variable.

**-P <port>, --port=<port>** Specifies the TCP port on which the server is listening for connections. Defaults to the value of the EDGEDB\_PORT environment variable or, if not set, to 5656.

**-u <username>, --user=<username>** Connect to the database as the user <username>. Defaults to the value of the EDGEDB\_USER environment variable, or, if not set, to the login name of the current OS user.

**-d <dbname>, --database=<dbname>** Specifies the name of the database to connect to. Default to the value of the EDGEDB\_DATABASE environment variable, or, if not set, to the calculated value of <username>.

**--password | --no-password** If --password is specified, force edgedb to prompt for a password before connecting to the database. This is usually not necessary, since edgedb will prompt for a password automatically if the server requires it.

Specifying --no-password disables all password prompts.

**--password-from-stdin** Use the first line of standard input as the password.

**--tls-ca-file /path/to/cert** Certificate to match server against.

This might either be full self-signed server certificate or certificate authority (CA) certificate that server certificate is signed with.

**--tls-security mode** Set the TLS security mode.

**default** Resolves to strict if no custom certificate is supplied via --tls-ca-file, environment variable, etc. Otherwise, resolves to no\_host\_verification.

**strict** Verify TLS certificate and hostname.

**no\_host\_verification** This allows using any certificate for any hostname. However, certificate must be present and match the root certificate specified with --tls-ca-file, credentials file, or system root certificates.

**insecure** Disable all TLS security measures.

**--wait-until-available=<wait\_time>** In case EdgeDB connection can't be established, keep retrying up to <wait\_time> (e.g. 30s).

**--connect-timeout=<timeout>** Specifies a <timeout> period. In case EdgeDB doesn't respond for this period the command will fail (or retry if --wait-until-available is also specified). The <timeout> value must be given using time units (e.g. hr, min, sec, ms, etc.). The default value is 10s.

## 7.2 Network usage

Generally command-line tools connect only to the database host. But there are two exceptions:

1. When the command-line tool starts, it checks if its version is up to date. *Details*
2. The *edgedb server* family of commands and *edgedb cli upgrade* discover package versions and docker images and also invoke package managers and the docker engine to do *index updates and related data*.

### 7.2.1 Version Check

Version check checks the current version of command-line tool by fetching [https://packages.edgedb.com/.jsonindexes/\\*.json](https://packages.edgedb.com/.jsonindexes/*.json).

Here is how such a request looks like:

```
GET /archive/.jsonindexes/linux-x86_64.json HTTP/1.1
host: packages.edgedb.com
content-length: 0
user-agent: edgedb
```

The User-Agent header only specifies that request is done by `edgedb` command-line tool (without version number). The platform, architecture and whether nightly is used can be devised from the URL of the query.

Latest version number is cached for the random duration from 16 to 32 hours (this randomization is done both for spreading the load and for better anonymizing the data). A failure is cached for the random duration from 6 to 12 hours.

### 7.2.2 Disabling Version Check

To disable version check do one of two things:

1. Use `--no-cli-update-check` command-line parameter to disable just for this invocation
2. Export `EDGEDB_RUN_VERSION_CHECK=never` in the environment.

To verify that check is skipped and no network access is being done logging facility can be used:

```
$ export RUST_LOG=edgedb::version_check=debug
$ edgedb --no-cli-update-check
[.snip..] Skipping version check due to --no-cli-update-check
edgedb>
$ EDGEDB_RUN_VERSION_CHECK=never edgedb
[.snip..] Skipping version check due to EDGEDB_RUN_VERSION_CHECK=never
edgedb>
```

### 7.2.3 edgedb server and edgedb self upgrade

Generally these commands do requests with exactly the headers like *version check*.

Data sources for the commands directly:

1. Package indexes and packages at <https://packages.edgedb.com>
2. Docker image index at <https://registry.hub.docker.com>

Data sources that can be used indirectly:

1. Docker engine may fetch indexes and images. Currently the only images used are at Docker Hub. More specifically are `edgedb/*` and `busybox` (Docker's official image).
2. Package managers (currently `apt-get`, `yum`) can fetch indexes and install packages from <https://packages.edgedb.com>. And as we use generic commands (e.g. `apt-get update`) and system dependencies, package manager can fetch package indexes and package data from any sources listed in repositories configured in the system.

To avoid reaching these hosts, avoid using: `edgedb server` and `edgedb self upgrade` subcommands. These commands only simplify installation and maintenance of the installations. All EdgeDB features are available without using them.

## 7.3 edgedb

**edb-alt-title** `edgedb` — Interactive Shell

EdgeDB interactive shell:

```
edgedb [<connection-option>...]
```

It's also possible to run an EdgeQL script by piping it into the EdgeDB shell. The shell will then run in non-interactive mode and print all the responses into the standard output:

```
cat myscript.edgeql | edgedb [<connection-option>...]
```

### 7.3.1 Description

`edgedb` is a terminal-based front-end to EdgeDB. It allows running queries and seeing results interactively.

### 7.3.2 Options

**-h, --help** Show help about the command and exit.

**--help-connect** Show all available *connection options*

**-V, --version** Print version.

**--no-cli-update-check** Disable version check.

**-I <name>, --instance=<name>** Specifies the named instance to connect to. The actual connection parameters are stored in `<edgedb_config_dir>/credentials` and are usually created by `edgedb instance create` or similar commands. Run `edgedb info` to see the location of `<edgedb_config_dir>` on your machine.

This option overrides host and port.

**--dsn=<dsn>** Specifies the DSN for EdgeDB to connect to.

This option overrides all other options except password.

**--credentials-file /path/to/file** Path to JSON file containing credentials.

**-H <hostname>, --host=<hostname>** Specifies the host name of the machine on which the server is running.  
Defaults to the value of the EDGEDB\_HOST environment variable.

**-P <port>, --port=<port>** Specifies the TCP port on which the server is listening for connections. Defaults to the value of the EDGEDB\_PORT environment variable or, if not set, to 5656.

**-u <username>, --user=<username>** Connect to the database as the user <username>. Defaults to the value of the EDGEDB\_USER environment variable, or, if not set, to the login name of the current OS user.

**-d <dbname>, --database=<dbname>** Specifies the name of the database to connect to. Default to the value of the EDGEDB\_DATABASE environment variable, or, if not set, to the calculated value of <username>.

**--password | --no-password** If --password is specified, force edgedb to prompt for a password before connecting to the database. This is usually not necessary, since edgedb will prompt for a password automatically if the server requires it.

Specifying --no-password disables all password prompts.

**--password-from-stdin** Use the first line of standard input as the password.

**--tls-ca-file /path/to/cert** Certificate to match server against.

This might either be full self-signed server certificate or certificate authority (CA) certificate that server certificate is signed with.

**--tls-security mode** Set the TLS security mode.

**default** Resolves to **strict** if no custom certificate is supplied via --tls-ca-file, environment variable, etc. Otherwise, resolves to **no\_host\_verification**.

**strict** Verify TLS certificate and hostname.

**no\_host\_verification** This allows using any certificate for any hostname. However, certificate must be present and match the root certificate specified with --tls-ca-file, credentials file, or system root certificates.

**insecure** Disable all TLS security measures.

**--wait-until-available=<wait\_time>** In case EdgeDB connection can't be established, keep retrying up to <wait\_time> (e.g. 30s).

**--connect-timeout=<timeout>** Specifies a <timeout> period. In case EdgeDB doesn't respond for this period the command will fail (or retry if --wait-until-available is also specified). The <timeout> value must be given using time units (e.g. hr, min, sec, ms, etc.). The default value is 10s.

### 7.3.3 Backslash Commands

#### Introspection

The introspection commands share a few common options that are available to many of the commands:

- **-v-** Verbose
- **-s-** Show system objects
- **-c-** Case-sensitive pattern matching

**\d [-v] OBJECT-NAME, \describe [-v] OBJECT-NAME** Describe schema object specified by *OBJECT-NAME*.

---

```
\ds, \d schema, \describe schema Describe the entire schema.  
\l, \list databases List databases.  
\ls [-sc] [PATTERN], \list scalars [-sc] [PATTERN] List scalar types.  
\lt [-sc] [PATTERN], \list types [-sc] [PATTERN] List object types.  
\lr [-c] [PATTERN], \list roles [-c] [PATTERN] List roles.  
\lm [-c] [PATTERN], \list modules [-c] [PATTERN] List modules.  
\la [-vsc] [PATTERN], \list aliases [-vsc] [PATTERN] List expression aliases.  
\lc [-c] [PATTERN], \list casts [-c] [PATTERN] List available conversions between types.  
\li [-vsc] [PATTERN], \list indexes [-vsc] [PATTERN] List indexes.
```

## Database

```
\database create [NAME] Create a new database.
```

## Query Analysis

```
\analyze QUERY
```

---

**Note:** This backslash command is compatible with EdgeDB server 3.0 and above.

---

Run a query performance analysis on the given query.

## Data Operations

```
\dump FILENAME Dump current database to a file at FILENAME.
```

```
\restore FILENAME Restore the database dump at FILENAME into the currently connected database.
```

## Editing

```
\s, \history Show a history of commands executed in the shell.
```

```
\e, \edit [N] Spawn $EDITOR to edit the most recent history entry or history entry N. History entries are negative indexed with -1 being the most recent command. Use the \history command (above) to see previous command indexes.
```

The output of this will then be used as input into the shell.

### Settings

**\set [OPTION [VALUE]]** If *VALUE* is omitted, the command will show the current value of *OPTION*. With *VALUE*, the option named by *OPTION* will be set to the provided value. Use `\set` with no arguments for a listing of all available options.

### Connection

**\c, \connect [DBNAME]** Connect to database *DBNAME*.

### Migrations

These migration commands are also accessible directly from the command line without first entering the EdgeDB shell. Their counterpart commands are noted and linked in their descriptions if you want more detail.

**\migration create** Create a migration script based on differences between the current database and the schema file, just like running `edgedb migration create`.

**\migrate, \migration apply** Apply your migration, just like running the `edgedb migrate`.

**\migration edit** Spawns \$EDITOR on the last migration file and fixes the migration ID after the editor exits, just like `edgedb migration edit`. This is typically used only on migrations that have not yet been applied.

**\migration log** Show the migration history, just like `edgedb migration log`.

**\migration status** Show how the state of the schema in the EdgeDB instance compares to the migration stored in the schema directory, just like `edgedb migration status`.

### Help

**\?, \h, \help** Show help on backslash commands.

**\q, \quit, \exit** Quit the REPL. You can also do this by pressing Ctrl+D.

## 7.4 edgedb dump

Backup an EdgeDB database to a file.

```
edgedb dump [<options>] <path>
```

### 7.4.1 Description

`edgedb dump` is a terminal command used to backup an EdgeDB database into a file.

## 7.4.2 Options

The `dump` command backups the database it is connected to. For specifying the connection target see [connection options](#).

**<path>** The name of the file to backup the database into.

**--all** Dump all databases and the server configuration using the directory specified by the `<path>`.

**--format=<format>** Choose dump format. For normal dumps this parameter should be omitted. For `--all` only `--format=dir` is required.

## 7.5 edgedb restore

Restore an EdgeDB database from a backup file.

```
edgedb restore [<options>] <path>
```

### 7.5.1 Description

`edgedb restore` is a terminal command used to restore an EdgeDB database from a backup file. An empty target database must be created before using this command.

### 7.5.2 Options

The `restore` command restores the backup file into the database it is connected to. For specifying the connection target see [connection options](#).

**<path>** The name of the backup file to restore the database from.

**--all** Restore all databases and the server configuration using the directory specified by the `<path>`.

**-v, --verbose** Verbose output.

## 7.6 edgedb configure

Configure the EdgeDB server.

```
edgedb configure [<connection-options>] <action> \
    [<parameter> <value>] \
    [<parameter-class> --<property>=<value> ...]
```

### 7.6.1 Description

`edgedb configure` is a terminal command used to alter the configuration of an EdgeDB instance. There are three types of configuration actions that can be performed.

### 7.6.2 Actions

**`edgedb configure insert`** Insert a new configuration entry for a setting that supports multiple configuration objects (e.g. Auth or Port).

**`edgedb configure set`** Set a scalar configuration value.

**`edgedb configure reset`** Reset an existing configuration entry or remove all values for an entry that supports multiple configuration objects.

### 7.6.3 Options

Most of the options are the same across all of the different configuration actions.

**<connection-options>** See [connection options](#).

**<parameter>** The name of a primitive configuration parameter. Available configuration parameters are described in the [Config](#) section.

**<value>** A value literal for a given configuration parameter or configuration object property.

**<parameter-class>** The name of a composite configuration value class. Available configuration classes are described in the [Config](#) section.

**--<property>=<value>** Set the <property> of a configuration object to <value>.

## 7.7 edgedb watch

---

**Note:** This CLI feature is compatible with EdgeDB server 3.0 and above.

---

Start a long-running process that watches for changes in schema files in your project's `dbschema` directory and applies those changes to your database in real time.

```
edgedb watch
```

---

**Note:** If a schema change cannot be applied, you will see an error in the `edgedb watch` console. You will also receive the error when you try to run a query with any EdgeDB client binding.

---

To learn about our recommended development migration workflow using `edgedb watch`, read our [intro to migrations](#).

## 7.8 edgedb migration

EdgeDB provides schema migration tools as server-side tools. This means that from the point of view of the application migrations are language- and platform-agnostic and don't require additional libraries.

Using the migration tools is the recommended way to make schema changes.

### 7.8.1 edgedb migration create

The next step after setting up the desired target schema is creating a migration script. This is done by invoking the following command:

```
edgedb migration create [<options>]
```

This will start an interactive tool that will provide the user with suggestions based on the differences between the current database and the schema file. The prompts will look something like this:

```
did you create object type 'default::User'? [y,n,l,c,b,s,q,?]
?

y - confirm the prompt, use the DDL statements
n - reject the prompt
l - list the DDL statements associated with prompt
c - list already confirmed EdgeQL statements
b - revert back to previous save point, perhaps previous question
s - stop and save changes (splits migration into multiple)
q - quit without saving changes
h or ? - print help
```

#### 7.8.1.1 Options

The `migration create` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**--non-interactive** Do not prompt user for input. By default this works only if all the changes are “safe” unless `--allow-unsafe` is also specified.

**--allow-unsafe** Apply the most probable unsafe changes in case there are any. This is only useful in non-interactive mode.

**--allow-empty** Create a new migration even if there are no changes. This is useful for creating migration stubs for data-only migrations.

**--schema-dir=<schema-dir>** Directory where the schema files are located. Defaults to `./dbschema`.

**--squash**

---

**Note:** This CLI feature is compatible with EdgeDB server 3.0 and above.

---

Squashes all your migrations into a single migration.

## 7.8.2 edgedb migration apply

Once the migration scripts are in place the changes can be applied to the database by this command:

```
edgedb migration apply [<options>]
```

The tool will find all the unapplied migrations in `dbschema/migrations/` directory and sequentially run them on the target instance.

### 7.8.2.1 Options

The `migration apply` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**--quiet** Do not print any messages, only indicate success by exit status.

**--schema-dir=<schema-dir>** Directory where the schema files are located. Defaults to `./dbschema`.

**--to-revision=<to-revision>** Upgrade to a specified revision.

Unique prefix of the revision can be specified instead of full revision name.

If this revision is applied, the command is no-op. The command ensures that this revision present, but it's not an error if more revisions are applied on top.

**--dev-mode**

---

**Note:** The `--dev-mode` option is compatible with EdgeDB server 3.0 and above.

---

Apply the current schema changes on top of the current migration history, without having created a new migration. This works the same way as `edgedb watch` but without starting a long-running watch task.

## 7.8.3 edgedb migration log

Show all migration versions.

```
edgedb migration log [<options>]
```

The tool will display the migration history either by reading it from the EdgeDB instance or from the schema directory.

### 7.8.3.1 Options

The `migration log` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**--from-fs** Print revisions from the schema directory (no database connection required). At least one of `--from-db` or `--from-fs` is required for `migration log` command.

**--from-db** Print revisions from the database (no schema files required). At least one of `--from-db` or `--from-fs` is required for `migration log` command.

**--newest-first** Sort migrations starting from newer to older, by default older revisions go first.

**--schema-dir=<schema-dir>** Directory where the schema files are located. Defaults to `./dbschema`.

**--limit=<N>** Show maximum of `N` revisions (default is unlimited).

## 7.8.4 edgedb migration status

Show current migration state.

```
edgedb migration status [<options>]
```

The tool will show how the state of the schema in the EdgeDB instance compares to the migrations stored in the schema directory.

### 7.8.4.1 Options

The `migration status` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**--quiet** Do not print any messages, only indicate success by exit status.

**--schema-dir=<schema-dir>** Directory where the schema files are located. Defaults to `./dbschema`.

## 7.8.5 edgedb migration edit

Edit migration file.

```
edgedb migration edit [<options>]
```

Invokes `$EDITOR` on the last migration file, and then fixes migration id after editor exits. Usually should be used for migrations that haven't been applied yet.

### 7.8.5.1 Options

The `migration edit` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**--schema-dir=<schema-dir>** Directory where the schema files are located. Defaults to `./dbschema`.

**--no-check** Do not check migration within the database connection.

**--non-interactive** Fix migration id non-interactively, and don't run editor.

## 7.8.6 edgedb migration upgrade-check

Checks your schema against the new EdgeDB version. You can add `--to-version <version>`, `--to-testing`, `--to-nightly`, or `--to-channel <channel>` to check against a specific version.

```
edgedb migration update-check [<options>]
```

### 7.8.6.1 Options

The `migration upgrade-check` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**--schema-dir=<schema-dir>** Directory where the schema files are located. Defaults to `./dbschema`.  
**--to-version <to\_version>** Check the upgrade to a specified version  
**--to-nightly** Check the upgrade to a latest nightly version  
**--to-testing** Check the upgrade to a latest testing version  
**--to-channel <to\_channel>** Check the upgrade to the latest version in the channel [possible values: stable, testing, nightly]  
**--watch** Monitor schema changes and check again on change

## 7.8.7 Setup

First of all, the migration tools need a place to store the schema and migration information. By default they will look in `dbschema` directory, but it's also possible to specify any other location by using `schema-dir` option. Inside this directory there should be an `.esdl` file with [SDL](#) schema description. It's also possible to split the schema definition across multiple `.esdl` files. The migration tools will read all of them and treat them as a single SDL document.

<code>edgedb migration create</code>	Create a migration script
<code>edgedb migration apply</code>	Bring current database to the latest or a specified revision
<code>edgedb migration log</code>	Show all migration versions
<code>edgedb migration status</code>	Show current migration state
<code>edgedb migration edit</code>	Edit migration file

## 7.9 edgedb migrate

This command is an alias for `edgedb migration apply`. Once the migration scripts are in place the changes can be applied to the database by this command.

## 7.10 edgedb database create

Create a new `database`.

```
edgedb database create [<options>] <name>
```

### 7.10.1 Description

`edgedb database create` is a terminal command equivalent to `create database`.

### 7.10.2 Options

The `database create` command runs in the EdgeDB instance it is connected to. For specifying the connection target see *connection options*.

**<name>** The name of the new database.

## 7.11 edgedb describe

The `edgedb describe` group of commands contains various schema introspection tools.

### 7.11.1 edgedb describe object

Describe a named schema object.

```
edgedb describe object [<options>] <name>
```

#### 7.11.1.1 Description

`edgedb describe` is a terminal command equivalent to `describe object` introspection command.

#### 7.11.1.2 Options

The `describe` command runs in the database it is connected to. For specifying the connection target see *connection options*.

**--verbose** This is equivalent to running `describe object ... as text verbose` command, which enables displaying additional details, such as annotations and constraints, which are otherwise omitted.

**<name>** Name of the schema object to describe.

### 7.11.2 edgedb describe schema

Give an *SDL* description of the schema of the database specified by the connection options.

```
edgedb describe schema [<options>]
```

### 7.11.2.1 Description

`edgedb describe schema` is a terminal command equivalent to `describe schema as sdl` introspection command.

### 7.11.2.2 Options

The `describe` command runs in the database it is connected to. For specifying the connection target see [connection options](#).

<code>edgedb describe object</code>	Describe a named schema object
<code>edgedb describe schema</code>	Describe schema of the current database

## 7.12 edgedb list

List matching database objects by name and type.

```
edgedb list <type> [<options>] <pattern>
```

### 7.12.1 Description

The `edgedb list` group of commands contains tools for listing database objects by matching name or type. The sub-commands are organized by the type of the objects listed.

### 7.12.2 Types

`edgedb list aliases` Display list of aliases defined in the schema.

`edgedb list casts` Display list of casts defined in the schema.

`edgedb list databases` Display list of databases in the server instance.

`edgedb list indexes` Display list of indexes defined in the schema.

`edgedb list modules` Display list of modules defined in the schema.

`edgedb list roles` Display list of roles in the server instance.

`edgedb list scalars` Display list of scalar types defined in the schema.

`edgedb list types` Display list of object types defined in the schema.

### 7.12.3 Options

The `list` command runs in the database it is connected to. For specifying the connection target see [connection options](#).

**-c, --case-sensitive** Indicates that the pattern should be treated in a case-sensitive manner.

**-s, --system** Indicates that built-in and objects should be included in the list.

**-v, --verbose** Include more details in the output.

**<pattern>** The pattern that the name should match. If omitted all objects of a particular type will be listed.

## 7.13 edgedb query

Execute one or more EdgeQL queries.

```
edgedb query [<options>] <edgeql-query>...
```

### 7.13.1 Description

`edgedb query` is a terminal command used to execute EdgeQL queries provided as space-separated strings.

### 7.13.2 Options

The `query` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**-F, --output-format=<output\_format>** Output format: `json`, `json-pretty`, `json-lines`, `tab-separated`. Default is `json-pretty`.

**-f, --file=<file>** Filename to execute queries from. Pass `--file -` to execute queries from stdin.

**<edgeql-query>** Any valid EdgeQL query to be executed.

## 7.14 edgedb analyze

---

**Note:** This CLI feature is compatible with EdgeDB server 3.0 and above.

---

Run a query performance analysis on the given query.

```
edgedb analyze [<options>] <query>
```

Here's example `analyze` output from a simple query:

```
Contexts
analyze select Hero {name, secret_identity, villains: {name, nemesis: {name}}}
Shape
— default::Hero (cost=20430.96)
    — .villains: default::Villain, default::Hero (cost=35.81)
```

### 7.14.1 Options

The `analyze` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

**<query>** The query to analyze. Be sure to wrap the query in quotes.

**--expand** Print expanded output of the query analysis

**--debug-output-file <debug\_output\_file>** Write analysis into the JSON file specified instead of formatting

**--read-json <read\_json>** Read JSON file instead of executing a query

## 7.15 edgedb ui

Open the EdgeDB UI of the current instance in your default browser.

```
edgedb ui [<options>]
```

### 7.15.1 Description

`edgedb ui` is a terminal command used to open the EdgeDB UI in your default browser. Alternatively, it can be used to print the UI URL with the `--print-url` option.

The EdgeDB UI is a tool that allows you to graphically manage and query your EdgeDB databases. It contains a REPL, a textual and graphical view of your database schemas, and a data explorer which allows for viewing your data as a table.

---

**Note:** The UI is served by default by development instances. To enable the UI on a production instance, use the `--admin-ui` option with `edgedb-server` or set the `EDGEDB_SERVER_ADMIN_UI` *environment variable* to enabled.

---

### 7.15.2 Options

The `ui` command runs on the database it is connected to. For specifying the connection target see [connection options](#).

- print-url** Print URL in console instead of opening in the browser. This is useful if you prefer to open the EdgeDB UI in a browser other than your default browser.
- no-server-check** Skip probing the UI endpoint of the server instance. The endpoint probe is in place to provide a friendly error if you try to connect to a UI on a remote instance that does not have the UI enabled.

## 7.16 edgedb info

Display information about the EdgeDB installation. Currently this command displays the filesystem paths used by EdgeDB.

```
edgedb info [<options>]
```

### 7.16.1 Paths

EdgeDB uses several directories, each storing different kinds of information. The exact path to these directories is determined by your operating system. Throughout the documentation, these paths are referred to as “EdgeDB config directory”, “EdgeDB data directory”, etc.

- **Config:** contains auto-generated credentials for all local instances and project metadata.
- **Data:** contains the *contents* of all local EdgeDB instances.
- **CLI Binary:** contains the CLI binary, if installed.
- **Service:** the home for running processes/daemons.
- **Cache:** a catchall for logs and various caches.

### 7.16.1.1 Options

**--get <path-name>** Return only a single path. <path-name> can be any of config-dir, cache-dir, data-dir, or service-dir.

## 7.17 edgedb project

EdgeDB provides a way to quickly setup a project. This way the project directory gets associated with a specific EdgeDB instance and thus makes it the default instance to connect to. This is done by creating an `edgedb.toml` file in the project directory.

### 7.17.1 edgedb project init

Setup a new project.

```
edgedb project init [<options>]
```

#### 7.17.1.1 Description

This command sets up a new project, creating an instance and a schema directory for it. It can also be used to convert an existing directory to a project directory, connecting the existing instance to the project. Typically this tool will prompt for specific details about how the project should be setup.

##### 7.17.1.1.1 EdgeDB Cloud

Users with access to the EdgeDB Cloud beta may use this command to create a Cloud instance after logging in using [edgedb cloud login](#).

To create a Cloud instance, your instance name should be in the format <github-username>/<instance-name>. Cloud instance names may contain alphanumeric characters and hyphens (i.e., -). You can provide this Cloud instance name through the interactive project initiation by running `edgedb project init` or by providing it via the `--server-instance` option.

#### 7.17.1.2 Options

**--link** Specifies whether the existing EdgeDB server instance should be linked with the project.

This option is useful for initializing a copy of a project freshly downloaded from a repository with a pre-existing project database.

**--no-migrations** Skip running migrations.

There are two main use cases for this option:

1. With `--link` option to connect to a datastore with existing data.
2. To initialize a new instance but then restore dump to it.

**--non-interactive** Run in non-interactive mode (accepting all defaults).

**--project-dir=<project-dir>** The project directory can be specified explicitly. Defaults to the current directory.

**--server-instance=<server-instance>** Specifies the EdgeDB server instance to be associated with the project.

**--server-version=<server-version>** Specifies the EdgeDB server instance to be associated with the project.

By default, when you specify a version, the CLI will use the latest release in the major version specified. This command, for example, will install the latest 2.x release:

```
$ edgedb project init --server-version 2.6
```

You may pin to a specific version by prepending the version number with an equals sign. This command will install version 2.6:

```
$ edgedb project init --server-version =2.6
```

---

**Note:** Some shells like ZSH may require you to escape the equals sign (e.g., \=2.6) or quote the version string (e.g., "=2.6").

---

### 7.17.2 edgedb project unlink

Remove association with and optionally destroy the linked EdgeDB instance.

```
edgedb project unlink [<options>]
```

#### 7.17.2.1 Description

This command unlinks the project directory from the instance. By default the EdgeDB instance remains untouched, but it can also be destroyed with an explicit option.

#### 7.17.2.2 Options

**-D, --destroy-server-instance** If specified, the associated EdgeDB instance is destroyed by running *edgedb instance destroy*.

**--non-interactive** Do not prompts user for input.

**--project-dir=<project-dir>** The project directory can be specified explicitly. Defaults to the current directory.

### 7.17.3 edgedb project info

Display various metadata about the project.

```
edgedb project info [OPTIONS]
```

### 7.17.3.1 Description

This command provides information about the project instance, such as name and the project path.

### 7.17.3.2 Options

**--instance-name** Display only the instance name.

**-j, --json** Output in JSON format.

**--project-dir=<project-dir>** The project directory can be specified explicitly. Defaults to the current directory.

## 7.17.4 edgedb project upgrade

Upgrade EdgeDB instance used for the current project

```
edgedb project upgrade [<options>]
```

### 7.17.4.1 Description

This command has two modes of operation.

- 1) Upgrade instance to a version specified in `edgedb.toml`. This happens when the command is invoked without any explicit target version.
- 2) Update `edgedb.toml` to a new version and upgrade the instance. Which happens when one of the options for providing the target version is used.

In all cases your data is preserved and converted using dump/restore mechanism. This might fail if lower version is specified (for example if upgrading from nightly to the stable version).

### 7.17.4.2 Options

**--force** Force upgrade process even if there is no new version.

**--to-latest** Upgrade to a latest stable version.

**--to-nightly** Upgrade to a latest nightly version.

**--to-version=<version>** Upgrade to a specified major version.

**--project-dir=<project-dir>** The project directory can be specified explicitly. Defaults to the current directory.

**-v, --verbose** Verbose output.

<code>edgedb project init</code>	Initialize a new or existing project
<code>edgedb project unlink</code>	Clean-up the project configuration
<code>edgedb project info</code>	Get various metadata about the project
<code>edgedb project upgrade</code>	Upgrade EdgeDB instance used for the current project

## 7.18 edgedb instance

The `edgedb instance` group of commands contains all sorts of tools for managing EdgeDB instances.

### 7.18.1 edgedb instance create

Initialize a new EdgeDB instance.

```
edgedb instance create [<options>] <name>
```

#### 7.18.1.1 Description

`edgedb instance create` is a terminal command for making a new EdgeDB instance and creating a corresponding credentials file in `<edgedb_config_dir>/credentials`. Run `edgedb info` to see the path to `<edgedb_config_dir>` on your machine.

##### 7.18.1.1.1 EdgeDB Cloud

Users with access to the EdgeDB Cloud beta may use this command to create a Cloud instance after logging in using [`edgedb cloud login`](#).

To create a Cloud instance, your instance name should be in the format `<github-username>/<instance-name>`. Cloud instance names may contain alphanumeric characters and hyphens (i.e., `-`).

#### 7.18.1.2 Options

`<name>` The new EdgeDB instance name.

`--nightly` Use the nightly server for this instance.

`--default-database=<default-database>` Specifies the default database name (created during initialization, and saved in credentials file). Defaults to `edgedb`.

`--default-user=<default-user>` Specifies the default user name (created during initialization, and saved in credentials file). Defaults to: `edgedb`.

`--port=<port>` Specifies which port should the instance be configured on. By default a random port will be used and recorded in the credentials file.

`--start-conf=<start-conf>` Configures how the new instance should start: `auto` for automatic start with the system or user session, `manual` to turn that off so that the instance can be manually started with [`edgedb instance start`](#) on demand. Defaults to: `auto`.

`--version=<version>` Specifies the version of the EdgeDB server to be used to run the new instance. To list the currently available options use [`edgedb server list-versions`](#).

By default, when you specify a version, the CLI will use the latest release in the major version specified. This command, for example, will install the latest 2.x release:

```
$ edgedb instance create --version 2.6 demo26
```

You may pin to a specific version by prepending the version number with an equals sign. This command will install version 2.6:

```
$ edgedb instance create --version =2.6 demo26
```

**Note:** Some shells like ZSH may require you to escape the equals sign (e.g., \=2.6) or quote the version string (e.g., "=2.6").

## 7.18.2 edgedb instance link

Authenticate a connection to a remote EdgeDB instance and assign an instance name to simplify future connections.

```
edgedb instance link [<options>] <name>
```

### 7.18.2.1 Description

`edgedb instance link` is a terminal command used to bind a set of connection credentials to an instance name. This is typically used as a way to simplify connecting to remote EdgeDB database instances. Usually there's no need to do this for local instances as `edgedb project init` will already set up a named instance.

### 7.18.2.2 Options

The `instance link` command uses the standard `connection options` for specifying the instance to be linked.

**<name>** Specifies a new instance name to associate with the connection options. If not present, the interactive mode will ask for the name.

**--non-interactive** Run in non-interactive mode (accepting all defaults).

**--quiet** Reduce command verbosity.

**--trust-tls-cert** Trust peer certificate.

**--overwrite** Overwrite existing credential file if any.

## 7.18.3 edgedb instance unlink

Unlink from a previously linked remote EdgeDB instance.

```
edgedb instance unlink <name>
```

### 7.18.3.1 Description

`edgedb instance unlink` is a terminal command used to unlink a remote instance. This removes the instance name from the list of valid instances.

### 7.18.3.2 Options

**<name>** Specifies the name of the remote instance to be unlinked.

## 7.18.4 edgedb instance list

Show all EdgeDB instances.

```
edgedb instance list [<options>]
```

### 7.18.4.1 Description

`edgedb instance list` is a terminal command that shows all the registered EdgeDB instances and some relevant information about them (status, port, etc.).

### 7.18.4.2 Options

**--extended** Output more debug info about each instance.

**-j, --json** Output in JSON format.

## 7.18.5 edgedb instance logs

Show instance logs.

```
edgedb instance logs [<options>] <name>
```

### 7.18.5.1 Description

`edgedb instance logs` is a terminal command for displaying the logs for a given EdgeDB instance.

### 7.18.5.2 Options

**<name>** The name of the EdgeDB instance.

**-n, --tail=<tail>** Number of the most recent lines to show.

**-f, --follow** Show log's tail and the continue watching for the new entries.

## 7.18.6 edgedb instance status

Show instance information.

```
edgedb instance status [<options>] [<name>]
```

### 7.18.6.1 Description

`edgedb instance status` is a terminal command for displaying the information about EdgeDB instances.

### 7.18.6.2 Options

**<name>** Show only the status of the specific EdgeDB instance.  
**--json** Format output as JSON.  
**--extended** Output more debug info about each instance.  
**--service** Show current systems service information.

## 7.18.7 edgedb instance start

Start an EdgeDB instance.

```
edgedb instance start [--foreground] <name>
```

### 7.18.7.1 Description

`edgedb instance start` is a terminal command for starting a new EdgeDB instance.

### 7.18.7.2 Options

**<name>** The EdgeDB instance name.  
**--foreground** Start the instance in the foreground rather than using systemd to manage the process (note you might need to stop non-foreground instance first).

## 7.18.8 edgedb instance stop

Stop an EdgeDB instance.

```
edgedb instance stop <name>
```

### 7.18.8.1 Description

`edgedb instance stop` is a terminal command for stopping a running EdgeDB instance. This is a necessary step before *destroying* an instance.

### 7.18.8.2 Options

**<name>** The EdgeDB instance name.

## 7.18.9 edgedb instance restart

Restart an EdgeDB instance.

```
edgedb instance restart <name>
```

### 7.18.9.1 Description

`edgedb instance restart` is a terminal command for restarting an EdgeDB instance.

### 7.18.9.2 Options

**<name>** The EdgeDB instance name.

## 7.18.10 edgedb instance destroy

Remove an EdgeDB instance.

```
edgedb instance destroy [<options>] <name>
```

### 7.18.10.1 Description

`edgedb instance destroy` is a terminal command for removing an EdgeDB instance and all its data.

### 7.18.10.2 Options

**<name>** The EdgeDB instance name.

**--force** Destroy the instance even if it is referred to by a project.

**-v, --verbose** Verbose output.

## 7.18.11 edgedb instance revert

Revert a major instance upgrade.

```
edgedb instance revert [<options>] <name>
```

### 7.18.11.1 Description

When `edgedb instance upgrade` performs a major version upgrade on an instance the old instance data is kept around. The `edgedb instance revert` command removes the new instance version and replaces it with the old copy. It also ensures that the previous version of EdgeDB server is used to run it.

### 7.18.11.2 Options

**<name>** The name of the EdgeDB instance to revert.

**--ignore-pid-check** Do not check if upgrade is in progress.

**-y, --no-confirm** Do not ask for a confirmation.

## 7.18.12 edgedb instance reset-password

Reset password for a user in the EdgeDB instance.

```
edgedb instance reset-password [<options>] <name>
```

### 7.18.12.1 Description

`edgedb instance reset-password` is a terminal command for resetting or updating the password for a user of an EdgeDB instance.

### 7.18.12.2 Options

**<name>** The name of the EdgeDB instance.

**--user=<user>** User to change password for. Defaults to the user in the credentials file.

**--password** Read the password from the terminal rather than generating a new one.

**--password-from-stdin** Read the password from stdin rather than generating a new one.

**--save-credentials** Save new user and password into a credentials file. By default credentials file is updated only if user name matches.

**--no-save-credentials** Do not save generated password into a credentials file even if user name matches.

**--quiet** Do not print any messages, only indicate success by exit status.

## 7.18.13 edgedb instance upgrade

Upgrade EdgeDB instance or installation.

```
edgedb instance upgrade [<options>] [<name>]
```

### 7.18.13.1 Description

This command is used to upgrade EdgeDB instances individually or in bulk.

### 7.18.13.2 Options

**<name>** The EdgeDB instance name to upgrade.  
**--force** Force upgrade process even if there is no new version.  
**--to-latest** Upgrade specified instance to the latest major version.  
**--to-nightly** Upgrade specified instance to a latest nightly version.  
**--local-minor** Upgrade all local instances to the latest minor versions.  
**--to-version=<version>** Upgrade to a specified major version.  
**-v, --verbose** Verbose output.

<i>edgedb instance create</i>	Initialize a new server instance
<i>edgedb instance link</i>	Link a remote instance
<i>edgedb instance unlink</i>	Unlink a remote instance
<i>edgedb instance list</i>	Show all instances
<i>edgedb instance logs</i>	Show logs of an instance
<i>edgedb instance status</i>	Show statuses of all or of a matching instance
<i>edgedb instance start</i>	Start an instance
<i>edgedb instance stop</i>	Stop an instance
<i>edgedb instance restart</i>	Restart an instance
<i>edgedb instance destroy</i>	Destroy a server instance and remove the data stored
<i>edgedb instance revert</i>	Revert a major instance upgrade
<i>edgedb instance reset-password</i>	Reset password for a user in the instance
<i>edgedb instance upgrade</i>	Upgrade installations and instances

## 7.19 edgedb server

The `edgedb server` group of commands contains all sorts of tools for managing EdgeDB server versions.

### 7.19.1 edgedb server info

Show server information.

```
edgedb server info [<options>]
```

### 7.19.1.1 Description

`edgedb server info` is a terminal command for displaying the information about installed EdgeDB servers.

### 7.19.1.2 Options

- json** Format output as JSON.
- nightly** Display the information about the nightly server version.
- latest** Display the information about the latest server version.
- bin-path** Display only the server binary path (if applicable).
- version=<version>** Display the information about a specific server version.

## 7.19.2 edgedb server install

Install EdgeDB server.

```
edgedb server install [<options>]
```

### 7.19.2.1 Description

`edgedb server install` is a terminal command for installing a specific EdgeDB server version.

### 7.19.2.2 Options

- i, --interactive** Performs the installation in interactive mode, similar to how [downloading and installing](#) works.
- nightly** Installs the nightly server version.
- version=<version>** Specifies the version of the server to be installed. Defaults to the most recent release.

## 7.19.3 edgedb server list-versions

List available and installed versions of the EdgeDB server.

```
edgedb server list-versions [<options>]
```

### 7.19.3.1 Description

`edgedb server list-versions` is a terminal command for displaying all the available EdgeDB server versions along with indicating whether or not and how they are currently installed.

### 7.19.3.2 Options

**--json** Format output as JSON.  
**--installed-only** Display only the installed versions.  
**--column=<column>** Format output as a single column displaying only one aspect of the server: `major-version`, `installed`, `available`.

## 7.19.4 `edgedb server uninstall`

Uninstall EdgeDB server.

```
edgedb server uninstall [<options>]
```

### 7.19.4.1 Description

`edgedb server uninstall` is a terminal command for removing a specific EdgeDB server version from your system.

### 7.19.4.2 Options

**--all** Uninstalls all server versions.  
**--nightly** Uninstalls the nightly server version.  
**--unused** Uninstalls server versions that are not used to run any instances.  
**--version=<version>** Specifies the version of the server to be uninstalled.  
**-v, --verbose** Produce a more verbose output.

<i>edgedb server info</i>	Show server information
<i>edgedb server install</i>	Install edgedb server
<i>edgedb server list-versions</i>	List available and installed versions of the server
<i>edgedb server uninstall</i>	Uninstall edgedb server

## 7.20 `edgedb cloud`

In addition to managing your own local and remote instances, the EdgeDB CLI offers tools to manage your instances running on our EdgeDB Cloud.

### 7.20.1 `edgedb cloud login`

Authenticate to the EdgeDB Cloud and remember the secret key locally

```
edgedb cloud login
```

This command will launch your browser and start the EdgeDB Cloud authentication flow. Once authentication is successful, the CLI will log a success message:

Successfully logged `in` to EdgeDB Cloud `as` <your\_email>

If you are unable to complete authentication in the browser, you can interrupt the command by pressing Ctrl-C.

---

**Note:** During the Cloud beta, you will only be able to successfully complete authentication if you have been invited to the beta.

---

## 7.20.2 edgedb cloud logout

Forget the stored access token

```
edgedb cloud logout [<options>]
```

### 7.20.2.1 Options

**--all-profiles** Logout from all Cloud profiles

**--force** Force log out from all profiles, even if linked to a project

**--non-interactive** Do not ask questions, assume user wants to log out of all profiles not linked to a project

## 7.20.3 edgedb cloud secretkey

Manage your secret keys

### 7.20.3.1 edgedb cloud secretkey create

Create a new secret key

```
edgedb cloud secretkey create [<options>]
```

---

**Note:** This command works only if you have already authenticated using `edgedb cloud login`.

---

### 7.20.3.1.1 Options

**--json** Output results as JSON

**-n, --name <name>** Friendly key name

**--description <description>** Long key description

**--expires <>duration> | "never"** Key expiration, in duration units, for example “1 hour 30 minutes”. If set to “never”, the key would not expire.

**--scopes <scopes>** Comma-separated list of key scopes. Mutually exclusive with `--inherit-scopes`.

**--inherit-scopes** Inherit key scopes from the currently used key. Mutually exclusive with `--scopes`.

**-y, --non-interactive** Do not ask questions, assume default answers to all inputs that have a default. Requires key TTL and scopes to be explicitly specified via `--ttl` or `--no-expiration`, and `--scopes` or `--inherit-scopes`.

### 7.20.3.2 `edgedb cloud secretkey list`

List existing secret keys

```
edgedb cloud secretkey list [<options>]
```

---

**Note:** This command works only if you have already authenticated using [`edgedb cloud login`](#).

---

#### 7.20.3.2.1 Options

**--json** Output results as JSON

### 7.20.3.3 `edgedb cloud secretkey revoke`

Revoke a secret key

```
edgedb cloud secretkey revoke [<options>] --secret-key-id <secret-key-id>
```

---

**Note:** This command works only if you have already authenticated using [`edgedb cloud login`](#).

---

#### 7.20.3.3.1 Options

**--json** Output results as JSON

**--secret-key-id <secret\_key\_id>** Id of secret key to revoke

**-y, --non-interactive** Revoke the key without asking for confirmation.

<a href="#"><code>edgedb cloud secretkey create</code></a>	Create a new secret key
<a href="#"><code>edgedb cloud secretkey list</code></a>	List existing secret keys
<a href="#"><code>edgedb cloud secretkey revoke</code></a>	Revoke a secret key

---

**Note:** These commands work only if you have already authenticated using [`edgedb cloud login`](#).

---

## 7.20.4 Usage

To use the CLI with EdgeDB Cloud, start by running `edgedb cloud login`. This will open a browser and allow you to log in to EdgeDB Cloud.

---

**Note:** During the Cloud beta, you will only be able to successfully complete authentication if you have been invited to the beta.

---

Once your login is complete, you may use the other CLI commands to create and interact with Cloud instances.

<code>edgedb cloud login</code>	Authenticate to the EdgeDB Cloud and remember the access token locally
<code>edgedb cloud logout</code>	Forget the stored access token
<code>edgedb cloud secretkey</code>	Manage your secret keys

## 7.21 edgedb cli upgrade

Upgrade the CLI binary.

```
edgedb cli upgrade [<options>]
```

### 7.21.1 Description

`edgedb cli upgrade` is a terminal command used to upgrade the CLI tools to keep them up-to-date.

### 7.21.2 Options

**--force** Reinstall the CLI even if there is no newer version.

**--quiet** Don't show the progress bar.

**--verbose** Enable verbose output.



## 8.1 EdgeQL

Statements in EdgeQL are a kind of an *expression* that has one or more `clauses` and is used to retrieve or modify data in a database.

Query statements:

- `select`

Retrieve data from a database and compute arbitrary expressions.

- `for`

Compute an expression for every element of an input set and concatenate the results.

- `group`

Group data into subsets by keys.

Data modification statements:

- `insert`

Create new object in a database.

- `update`

Update objects in a database.

- `delete`

Remove objects from a database.

Transaction control statements:

- `start transaction`

Start a transaction.

- `commit`

Commit the current transaction.

- `rollback`

Abort the current transaction.

- `declare savepoint`

Declare a savepoint within the current transaction.

- *rollback to savepoint*

Rollback to a savepoint within the current transaction.

- *release savepoint*

Release a previously declared savepoint.

Session state control statements:

- *set* and *reset*.

Introspection command:

- *describe*.

## 8.1.1 Lexical structure

Every EdgeQL command is composed of a sequence of *tokens*, terminated by a semicolon (;). The types of valid tokens as well as their order is determined by the syntax of the particular command.

EdgeQL is case sensitive except for *keywords* (in the examples the keywords are written in upper case as a matter of convention).

There are several kinds of tokens: *keywords*, *identifiers*, *literals* (constants) and *symbols* (operators and punctuation).

Tokens are normally separated by whitespace (space, tab, newline) or comments.

### 8.1.1.1 Identifiers

There are two ways of writing identifiers in EdgeQL: plain and quoted. The plain identifiers are similar to many other languages, they are alphanumeric with underscores and cannot start with a digit. The quoted identifiers start and end with a backtick `quoted.identifier` and can contain any characters inside with a few exceptions. They must not start with an ampersand (@) or contain a double colon (: :). If there's a need to include a backtick character as part of the identifier name a double-backtick sequence (` ``) should be used: `quoted``identifier` will result in the actual identifier being quoted`identifier`.

```
identifier      ::= plain_ident | quoted_ident
plain_ident    ::= ident_first ident_rest*
ident_first    ::= <any letter, underscore>
ident_rest     ::= <any letter, digits, underscore>
quoted_ident   ::= ` ` qident_first qident_rest* ` `
qident_first   ::= <any character except "@">
qident_rest    ::= <any character>
```

Quoted identifiers are usually needed to represent module names that contain a dot (.) or to distinguish *names* from *reserved keywords* (for instance to allow referring to a link named “order” as `order`).

### 8.1.1.2 Names and keywords

There are a number of *reserved* and *unreserved* keywords in EdgeQL. Every identifier that is not a *reserved* keyword is a valid *name*. *Names* are used to refer to concepts, links, link properties, etc.

```

short_name      ::=  not_keyword_ident | quoted_ident
not_keyword_ident ::= <any plain_ident except for keyword>
keyword         ::=  reserved_keyword | unreserved_keyword
reserved_keyword ::= case insensitive sequence matching any
                     of the following
                     "AGGREGATE" | "ALTER" | "AND" |
                     "ANY" | "COMMIT" | "CREATE" |
                     "DELETE" | "DETACHED" | "DISTINCT" |
                     "DROP" | "ELSE" | "EMPTY" | "EXISTS" |
                     "FALSE" | "FILTER" | "FUNCTION" |
                     "GET" | "GROUP" | "IF" | "ILIKE" |
                     "IN" | "INSERT" | "IS" | "LIKE" |
                     "LIMIT" | "MODULE" | "NOT" | "OFFSET" |
                     "OR" | "ORDER" | "OVER" |
                     "PARTITION" | "ROLLBACK" | "SELECT" |
                     "SET" | "SINGLETON" | "START" | "TRUE" |
                     "UPDATE" | "UNION" | "WITH"
unreserved_keyword ::= case insensitive sequence matching any
                     of the following
                     "ABSTRACT" | "ACTION" | "AFTER" |
                     "ARRAY" | "AS" | "ASC" | "ATOM" |
                     "ANNOTATION" | "BEFORE" | "BY" |
                     "CONCEPT" | "CONSTRAINT" |
                     "DATABASE" | "DESC" | "EVENT" |
                     "EXTENDING" | "FINAL" | "FIRST" |
                     "FOR" | "FROM" | "INDEX" |
                     "INITIAL" | "LAST" | "LINK" |
                     "MAP" | "MIGRATION" | "OF" | "ON" |
                     "POLICY" | "PROPERTY" |
                     "REQUIRED" | "RENAME" | "TARGET" |
                     "THEN" | "TO" | "TRANSACTION" |
                     "TUPLE" | "VALUE" | "VIEW"

```

Fully-qualified names consist of a module, `:::`, and a short name. They can be used in most places where a short name can appear (such as paths and shapes).

```

name      ::=  short_name | fq_name
fq_name   ::=  short_name ":" short_name |
                short_name ":" unreserved_keyword

```

### 8.1.1.3 Constants

A number of scalar types have literal constant expressions.

#### 8.1.1.3.1 Strings

Production rules for `str` literals:

```

string      ::= str | raw_str
str         ::= "" str_content* "" | ''' str_content* '''
raw_str     ::= "r'" raw_content* "'"
              | 'r'" raw_content* "'"
              | dollar_quote raw_content* dollar_quote
raw_content ::= <any character different from delimiting quote>
dollar_quote ::= $" q_char0? q_char* "$"
q_char0    ::= "A"..."Z" | "a"..."z" | "_"
q_char     ::= "A"..."Z" | "a"..."z" | "_" | "\0"..."9"
str_content ::= <newline> | unicode | str_escapes
unicode    ::= <any printable unicode character not preceded by "\">
str_escapes ::= <see below for details>

```

The inclusion of “high ASCII” character in `edgeql:q_char` in practice reflects the ability to use some of the letters with diacritics like ò or ü in the dollar-quote delimiter.

Here’s a list of valid `edgeql:str_escapes`:

Escape Sequence	Meaning
\[newline]	Backslash and all whitespace up to next non-whitespace character is ignored
\\	Backslash ()
\'	Single quote (‘)
\"	Double quote (“”)
\b	ASCII backspace (\x08)
\f	ASCII form feed (\x0C)
\n	ASCII newline (\x0A)
\r	ASCII carriage return (\x0D)
\t	ASCII tabulation (\x09)
\xhh	Character with hex value hh
\uhhhh	Character with 16-bit hex value hhhh
\Uhhhhhhhh	Character with 32-bit hex value hhhhhhhh

Here’s some examples of regular strings using escape sequences

```

db> select 'hello
... world';
{'hello
world'}
```

```

db> select "hello\nworld";
{'hello
world'}
```

(continues on next page)

(continued from previous page)

```
db> select 'hello \
...      world';
{'hello world'}
```

```
db> select 'https://edgedb.com/\
...      docs/edgeql/lexical\
...      #constants';
{'https://edgedb.com/docs/edgeql/lexical#constants'}
```

```
db> select 'hello \\ world';
{'hello \\ world'}
```

```
db> select 'hello \'world\'';
{"hello 'world'"}
```

```
db> select 'hello \x77orlд';
{'hello world'}
```

```
db> select 'hello \u0077orlд';
{'hello world'}
```

Raw strings don't have any specially interpreted symbols; they contain all the symbols between the quotes exactly as typed.

```
db> select r'hello \\ world';
{'hello \\ world'}
```

```
db> select r'hello \
... world';
{'hello \
world'}
```

```
db> select r'hello
... world';
{'hello
world'}
```

## Dollar-quoted String Constants

A special case of raw strings are *dollar-quoted* strings. They allow using either kind of quote symbols ' or " as part of the string content without the quotes terminating the string. In fact, because the *dollar-quote* delimiter sequences can have arbitrary alphanumeric additional fillers, it is always possible to surround any content with *dollar-quotes* in an unambiguous manner:

```
db> select $$hello
... world$$;
{'hello
world'}
```

```
db> select $$hello\nworld$$;
{'hello\nworld'}
```

(continues on next page)

(continued from previous page)

```
db> select $$"hello" 'world$$;
{\"hello\" 'world'}
```

```
db> select $a$hello$$world$$a$;
{'hello$world$'}
```

More specifically, a delimiter:

- Must start with an ASCII letter or underscore
- Has following characters that can be digits 0-9, underscores or ASCII letters

### 8.1.1.3.2 Bytes

Production rules for `bytes` literals:

```
bytes      ::=  "b'" bytes_content* "'" | 'b'' bytes_content* "'"
bytes_content ::=  <newline> | ascii | bytes_escapes
ascii      ::=  <any printable ascii character not preceded by "\">
bytes_escapes ::=  <see below for details>
```

Here's a list of valid `edgeql:bytes_escapes`:

Escape Sequence	Meaning
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\b	ASCII backspace (\x08)
\f	ASCII form feed (\x0C)
\n	ASCII newline (\x0A)
\r	ASCII carriage return (\x0D)
\t	ASCII tabulation (\x09)
\xhh	Character with hex value hh

### 8.1.1.3.3 Integers

There are two kinds of integer constants: limited size (`int64`) and unlimited size (`bigint`). Unlimited size integer `bigint` literals are similar to a regular integer literals with an `n` suffix. The production rules are as follows:

```
bigint    ::=  integer "n"
integer   ::=  "0" | non_zero digit*
non_zero  ::=  "1"..."9"
digit     ::=  "0"..."9"
```

By default all integer literals are interpreted as `int64`, while an explicit cast can be used to convert them to `int16` or `int32`:

```
db> select 0;
{0}

db> select 123;
{123}

db> select <int16>456;
{456}

db> select <int32>789;
{789}
```

Examples of *bigint* literals:

```
db> select 123n;
{123n}

db> select 12345678901234567890n;
{12345678901234567890n}
```

#### 8.1.1.3.4 Real Numbers

Just as for integers, there are two kinds of real number constants: limited precision (*float64*) and unlimited precision (*decimal*). The *decimal* constants have the same lexical structure as *float64*, but with an `n` suffix:

```
decimal      ::= float "n"
float        ::= float_wo_dec | float_w_dec
float_wo_dec ::= integer_part exp
float_w_dec  ::= integer_part "." decimal_part? exp?
integer_part ::= "0" | non_zero digit*
decimal_part ::= digit+
exp          ::= "e" ("+" | "-")? digit+
```

By default all float literals are interpreted as *float64*, while an explicit cast can be used to convert them to *float32*:

```
db> select 0.1;
{0.1}

db> select 12.3;
{12.3}

db> select 1e3;
{1000.0}

db> select 1.2e-3;
{0.0012}

db> select <float32>12.3;
{12.3}
```

Examples of *decimal* literals:

```
db> select 12.3n;  
{12.3n}  
  
db> select 12345678901234567890.12345678901234567890n;  
{12345678901234567890.12345678901234567890n}  
  
db> select 12345678901234567890.12345678901234567890e-3n;  
{12345678901234567.89012345678901234567890n}
```

#### 8.1.1.4 Punctuation

EdgeQL uses ; as a statement separator. It is idempotent, so multiple repetitions of ; don't have any additional effect.

#### 8.1.1.5 Comments

Comments start with a # character that is not otherwise part of a string literal and end at the end of line. Semantically, a comment is equivalent to whitespace.

```
comment ::= "#" <any other characters until the end of line>
```

#### 8.1.1.6 Operators

EdgeQL operators listed in order of precedence from lowest to highest:

operator
<i>union</i>
<i>if..else</i>
<i>or</i>
<i>and</i>
<i>not</i>
<i>=, !=, ?=, ?!=</i>
<i>&lt;, &gt;, &lt;=, &gt;=</i>
<i>like, ilike</i>
<i>in, not in</i>
<i>is, is not</i>
<i>+, -, ++</i>
<i>*, /, //, %</i>
<i>??</i>
<i>distinct, unary -</i>
<i>^</i>
<i>type cast</i>
<i>array[], str[], json[], bytes[]</i>
<i>detached</i>

## 8.1.2 Evaluation algorithm

EdgeQL is a functional language in the sense that every expression is a composition of one or more queries.

Queries can be *explicit*, such as a `select` statement, or *implicit*, as dictated by the semantics of a function, operator or a statement clause.

An implicit `select` subquery is assumed in the following situations:

- expressions passed as an argument for an aggregate function parameter or operand;
- the right side of the assignment operator (`:=`) in expression aliases and *shape element declarations*;
- the majority of statement clauses.

A nested query is called a *subquery*. Here, the phrase “*appearing directly in the query*” means “appearing directly in the query rather than in the subqueries”.

A query is evaluated recursively using the following procedure:

1. Make a list of simple paths appearing directly in the query. For every path in the list, find all paths which begin with the same set reference and treat their longest common prefix as an equivalent set reference.

Example:

```
select (
    User.firstname,
    User.friends.firstname,
    User.friends.lastname,
    Issue.priority.name,
    Issue.number,
    Status.name
);
```

In the above query, the longest common prefixes are: `User`, `User.friends`, `Issue`, and `Status.name`.

2. Make a *query input list* of all unique set references which appear directly in the query (including the common path prefixes identified above). The set references and path prefixes in this list are called *input set references*, and the sets they represent are called *input sets*. Order this list such that any input references come before any other input set reference for which it is a prefix (sorting lexicographically works).
3. Compute a set of *input tuples*.

- Begin with a set containing a single empty tuple.
- For each input set reference, we compute a *dependent* Cartesian product of the input tuple set (`X`) so far and the input set `Y` being considered. In this dependent product, we pair each tuple `x` in the input tuple set `X` with each element of the subset of the input set `Y` corresponding to the tuple `x`. (For example, in the above example, computing the dependent product of `User` and `User.friends` would pair each user with all of their friends.)

(Mathematically,  $X' = \{(x, y) \mid x \in X, y \in f(x)\}$ , if  $f(x)$  selects the appropriate subset.)

The set produced becomes the new input tuple set and we continue down the list.

- As a caveat to the above, if an input set appears exclusively as an *optional* argument, it produces pairs with a placeholder value `Missing` instead of an empty Cartesian product in the above set. (Mathematically, this corresponds to having  $f(x) = \{\text{Missing}\}$  whenever it would otherwise produce an empty set.)

4. Iterate over the set of input tuples, and on every iteration:

- in the query and its subqueries, replace each input set reference with the corresponding value from the input tuple or an empty set if the value is `Missing`;

- evaluate the query expression in the order of precedence using the following rules:
    - subqueries are evaluated recursively from step 1;
    - a function or an operator is evaluated in a loop over a Cartesian product of its non-aggregate arguments (empty `optional` arguments are excluded from the product); aggregate arguments are passed as a whole set; the results of the invocations are collected to form a single set.
5. Collect the results of all iterations to obtain the final result set.

### 8.1.3 Shapes

A *shape* is a powerful syntactic construct that can be used to describe type variants in queries, data in `insert` and `update` statements, and to specify the format of statement output.

Shapes always follow an expression, and are a list of *shape elements* enclosed in curly braces:

```
<expr> "{"
    <shape_element> [, ...]
"}"
```

Shape element has the following syntax:

```
[ "[" is <object-type> "]" ] <pointer-spec>
```

If an optional `<object-type>` filter is used, `<pointer-spec>` will only apply to those objects in the `<expr>` set that are instances of `<object-type>`.

`<pointer-spec>` is one of the following:

- a name of an existing link or property of a type produced by `<expr>`;
- a declaration of a computed link or property in the form

```
[@]<name> := <ptrexpr>
```

- a *subshape* in the form

```
<pointer-name>: [ "[" is <target-type> "]" ] "{" ... "}"`
```

The `<pointer-name>` is the name of an existing link or property, and `<target-type>` is an optional object type that specifies the type of target objects selected or inserted, depending on the context.

#### 8.1.3.1 Shaping Query Results

At the end of the day, EdgeQL has two jobs that are similar, yet distinct:

- 1) Express the values that we want computed.
- 2) Arrange the values into a particular shape that we want.

Consider the task of getting “names of users and all of the friends’ names associated with the given user” in a database defined by the following schema:

```
type User {
    required property name -> str;
    multi link friends -> User;
}
```

```
type User {
    required name: str;
    multi friends: User;
}
```

If we only concern ourselves with getting the values, then a reasonable solution to this might be:

```
db> select (User.name, User.friends.name ?? '');
{
    ('Alice', 'Cameron'),
    ('Alice', 'Dana'),
    ('Billie', 'Dana'),
    ('Cameron', ''),
    ('Dana', 'Alice'),
    ('Dana', 'Billie'),
    ('Dana', 'Cameron'),
}
```

This particular solution is very similar to what one might get using SQL. It's equivalent to a table with "user name" and "friend name" columns. It gets the job done, albeit with some redundant repeating of "user names".

We can improve things a little and reduce the repetition by aggregating all the friend names into an array:

```
db> select (User.name, array_agg(User.friends.name));
{
    ('Alice', ['Cameron', 'Dana']),
    ('Billie', ['Dana']),
    ('Cameron', []),
    ('Dana', ['Alice', 'Billie', 'Cameron']),
}
```

This achieves a couple of things: it's easier to see which friends belong to which user and we no longer need the placeholder '' for those users who don't have friends.

The recommended way to get this information in EdgeDB, however, is to use *shapes*, because they mimic the structure of the data and the output:

```
db> select User {
...     name,
...     friends: {
...         name
...     }
... };
{
    default::User {
        name: 'Alice',
        friends: {
            default::User {name: 'Cameron'},
            default::User {name: 'Dana'},
        },
    },
    default::User {name: 'Billie', friends: {default::User {name: 'Dana'}}},
    default::User {name: 'Cameron', friends: {}},
    default::User {
```

(continues on next page)

(continued from previous page)

```

name: 'Dana',
friends: {
    default::User {name: 'Alice'},
    default::User {name: 'Billie'},
    default::User {name: 'Cameron'},
},
},
}

```

So far the expression for the data that we wanted was also acceptable for structuring the output, but what if that's not the case? Let's add a condition and only show those users who have friends with either the letter "i" or "o" in their names:

```

db> select User {
...     name,
...     friends: {
...         name
...
...     }
... } filter .friends.name ilike '%i%' or .friends.name ilike '%o%';
{
    default::User {
        name: 'Alice',
        friends: {
            default::User {name: 'Cameron'},
            default::User {name: 'Dana'},
        },
    },
    default::User {
        name: 'Dana',
        friends: {
            default::User {name: 'Alice'},
            default::User {name: 'Billie'},
            default::User {name: 'Cameron'},
        },
    },
}

```

That `filter` is getting a bit bulky, so perhaps we can just factor these flags out as part of the shape's computed properties:

```

db> select User {
...     name,
...     friends: {
...         name
...
...     },
...     has_i := .friends.name ilike '%i%',
...     has_o := .friends.name ilike '%o%',
... } filter .has_i or .has_o;
{
    default::User {
        name: 'Alice',
        friends: {

```

(continues on next page)

(continued from previous page)

```

default::User {name: 'Cameron'},
default::User {name: 'Dana'},
},
has_i: {false, false},
has_o: {true, false},
},
default::User {
  name: 'Dana',
  friends: {
    default::User {name: 'Alice'},
    default::User {name: 'Billie'},
    default::User {name: 'Cameron'},
  },
  has_i: {true, true, false},
  has_o: {false, false, true},
},
}

```

It looks like this refactoring came at the cost of putting extra things into the output. In this case we don't want our intermediate calculations to actually show up in the output, so what can we do? In EdgeDB the output structure is determined *only* by the expression appearing in the top-level `select`. This means that we can move our intermediate calculations into the `with` block:

```

db> with U := (
...     select User {
...         has_i := .friends.name ilike '%i%',
...         has_o := .friends.name ilike '%o%',
...     }
... )
... select U {
...     name,
...     friends: {
...         name
...     },
... } filter .has_i or .has_o;
{
  default::User {
    name: 'Alice',
    friends: {
      default::User {name: 'Cameron'},
      default::User {name: 'Dana'},
    },
  },
  default::User {
    name: 'Dana',
    friends: {
      default::User {name: 'Alice'},
      default::User {name: 'Billie'},
      default::User {name: 'Cameron'},
    },
  },
}

```

This way we can use `has_i` and `has_o` in our query without leaking them into the output.

### 8.1.3.2 General Shaping Rules

In EdgeDB typically all shapes appearing in the top-level `select` should be reflected in the output. This also applies to shapes no matter where and how they are nested. Aside from other shapes, this includes nesting in arrays:

```
db> select array_agg(User {name});  
{  
    [  
        default::User {name: 'Alice'},  
        default::User {name: 'Billie'},  
        default::User {name: 'Cameron'},  
        default::User {name: 'Dana'},  
    ],  
}
```

... or tuples:

```
db> select enumerate(User {name});  
{  
    (0, default::User {name: 'Alice'}),  
    (1, default::User {name: 'Billie'}),  
    (2, default::User {name: 'Cameron'}),  
    (3, default::User {name: 'Dana'}),  
}
```

You can safely access a tuple element and expect the output shape to be intact:

```
db> select enumerate(User{name}).1;  
{  
    default::User {name: 'Alice'},  
    default::User {name: 'Billie'},  
    default::User {name: 'Cameron'},  
    default::User {name: 'Dana'},  
}
```

Accessing array elements or working with slices also preserves output shape and is analogous to using `offset` and `limit` when working with sets:

```
db> select array_agg(User {name})[2];  
{default::User {name: 'Cameron'}}
```

### 8.1.3.3 Losing Shapes

There are some situations where shape information gets completely or partially discarded. Any such operation also prevents the altered shape from appearing in the output altogether.

In order for the shape to be preserved, the original expression type must be preserved. This means that `union` can alter the shape, because the result of a `union` is a `union type`. So you can still refer to the common properties, but not to the properties that appeared in the shape.

As mentioned above, since `union` potentially alters the expression shape it never preserves output shape, even when the underlying type wasn't altered:

```
db> select User{name} union User{name};
{
    default::User {id: 7769045a-27bf-11ec-94ea-3f6c0ae59eb3},
    default::User {id: 7b42ed20-27bf-11ec-94ea-7700ec77834e},
    default::User {id: 7fcedb4-27bf-11ec-94ea-73dcb6f297a4},
    default::User {id: 82f52646-27bf-11ec-94ea-3718ffb8dd15},
    default::User {id: 7769045a-27bf-11ec-94ea-3f6c0ae59eb3},
    default::User {id: 7b42ed20-27bf-11ec-94ea-7700ec77834e},
    default::User {id: 7fcedb4-27bf-11ec-94ea-73dcb6f297a4},
    default::User {id: 82f52646-27bf-11ec-94ea-3718ffb8dd15},
}
```

Listing several items inside a set { ... } functions identically to a `union` and so will also produce a union type and remove shape from output.

Another subtle way for a type union to remove the shape from the output is by the `??` and the `if..else` operators. Both of them determine the result type as the union of the left and right operands:

```
db> select <User>{} ?? User {name};
{
    default::User {id: 7769045a-27bf-11ec-94ea-3f6c0ae59eb3},
    default::User {id: 7b42ed20-27bf-11ec-94ea-7700ec77834e},
    default::User {id: 7fcedb4-27bf-11ec-94ea-73dcb6f297a4},
    default::User {id: 82f52646-27bf-11ec-94ea-3718ffb8dd15},
```

Shapes survive array creation (either via `array_agg()` or by using [ ... ]), but they follow the same rules as for `union` for array `concatenation`. Basically the element type of the resulting array must be a union type and thus all shape information is lost:

```
db> select array_agg(User{name}) ++ array_agg(User{name});
{
    [
        default::User {id: 7769045a-27bf-11ec-94ea-3f6c0ae59eb3},
        default::User {id: 7b42ed20-27bf-11ec-94ea-7700ec77834e},
        default::User {id: 7fcedb4-27bf-11ec-94ea-73dcb6f297a4},
        default::User {id: 82f52646-27bf-11ec-94ea-3718ffb8dd15},
        default::User {id: 7769045a-27bf-11ec-94ea-3f6c0ae59eb3},
        default::User {id: 7b42ed20-27bf-11ec-94ea-7700ec77834e},
        default::User {id: 7fcedb4-27bf-11ec-94ea-73dcb6f297a4},
        default::User {id: 82f52646-27bf-11ec-94ea-3718ffb8dd15},
    ],
}
```

---

**Note:** The `for` statement preserves the shape given inside the `union` clause, effectively applying the shape to its entire result.

---

## 8.1.4 Paths

A *path expression* (or simply a *path*) represents a set of values that are reachable when traversing a given sequence of links or properties from some source set.

The result of a path expression depends on whether it terminates with a link or property reference.

- a) if a path *does not* end with a property reference, then it represents a unique set of objects reachable from the set at the root of the path;
- b) if a path *does* end with a property reference, then it represents a list of property values for every element in the unique set of objects reachable from the set at the root of the path.

The syntactic form of a path is:

```
<expression> <path-step> [ <path-step> ... ]  
  
# where <path-step> is:  
  <step-direction> <pointer-name>
```

The individual path components are:

**<expression>** Any valid expression.

**<step-direction>** It can be one of the following:

- . for an outgoing link reference
- .< for an incoming or *backlink* reference
- @ for a link property reference

**<pointer-name>** This must be a valid link or link property name.

## 8.1.5 Casts

There are different ways that casts appear in EdgeQL.

### 8.1.5.1 Explicit Casts

A type cast expression converts the specified value to another value of the specified type:

```
"<" <type> ">" <expression>
```

The **<type>** must be a valid *type expression* denoting a non-abstract scalar or a container type.

For example, the following expression casts an integer value into a string:

```
db> select <str>10;  
{"10"}
```

See the *type cast operator* section for more information on type casting rules.

You can cast a UUID into an object:

```
db> select <Hero><uuid>'01d9cc22-b776-11ed-8bef-73f84c7e91e7';  
{default::Hero {id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7}}
```

If you try to cast a UUID that no object of the type has as its `id` property, you'll get an error:

```
db> select <Hero><uuid>'aaaaaaaa-aaaa-aaa-aaaa-aaaaaaaaaa';
edgedb error: CardinalityViolationError: 'default::Hero' with id 'aaaaaaaa-aaaa-aaa-
˓→aaaa-aaaaaaaaaa' does not exist
```

### 8.1.5.2 Assignment Casts

*Assignment casts* happen when inserting new objects. Numeric types will often be automatically cast into the specific type corresponding to the property they are assigned to. This is to avoid extra typing when dealing with numeric value using fewer bits:

```
# Automatically cast a literal 42 (which is int64
# by default) into an int16 value.
insert MyObject {
    int16_val := 42
};
```

If *assignment* casting is supported for a given pair of types, *explicit* casting of those types is also supported.

### 8.1.5.3 Implicit Casts

*Implicit casts* happen automatically whenever the value type doesn't match the expected type in an expression. This is mostly supported for numeric casts that don't incur any potential information loss (in form of truncation), so typically from a less precise type, to a more precise one. The `int64` to `float64` is a notable exception, which can suffer from truncation of significant digits for very large integer values. There are a few scenarios when *implicit casts* can occur:

- 1) Passing arguments that don't match exactly the types in the function signature:

```
db> with x := <float32>12.34
... select math::ceil(x);
{13}
```

The function `math::ceil()` only takes `int64`, `float64`, `bigint`, or `decimal` as its argument. So the `float32` value will be *implicitly cast* into a `float64` in order to match a valid signature.

- 2) Using operands that don't match exactly the types in the operator signature (this works the same way as for functions):

```
db> select 1 + 2.3;
{3.3}
```

The operator `+` is defined only for operands of the same type, so in the expression above the `int64` value `1` is *implicitly cast* into a `float64` in order to match the other operand and produce a valid signature.

- 3) Mixing different numeric types in a set:

```
db> select {1, 2.3, <float32>4.5} is float64;
{true, true, true}
```

All elements in a set have to be of the same type, so the values are cast into `float64` as that happens to be the common type to which all the set elements can be *implicitly cast*. This would work out the same way if `union` was used instead:

```
db> select (1 union 2.3 union <float32>4.5) is float64;
{true, true, true}
```

If *implicit* casting is supported for a given pair of types, *assignment* and *explicit* casting of those types is also supported.

#### 8.1.5.4 Casting Table

**Note:** The UUID-to-object cast is only available in EdgeDB 3.0+.

from \ to	json	str	float32	float64	int16	int32	int64	bigint	decimal	bool	bytes	uuid	date	duration	localdate	localtime	object
<i>json</i>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	
<i>str</i>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	<>	
<i>float32</i>	<>		impl	<>	<>*	<>*	<>	<>									
<i>float64</i>	<>	:=		<>	<>*	<>*	<>	<>									
<i>int16</i>	<>	impl	impl		impl	impl	impl	impl	impl								
<i>int32</i>	<>			impl	<>		impl	impl	impl	impl							
<i>int64</i>	<>	:=	impl	:=	:=			impl	impl								
<i>bigint</i>								impl									
<i>decimal</i>	<>	<>	<>	<>	<>	<>	<>	<>									
<i>bool</i>	<>															<>	
<i>bytes</i>																	
<i>uuid</i>	<>															<>	
<i>datetime</i>																	
<i>duration</i>																	
<i>localdate</i>																<>	
<i>localdatetime</i>															<>	<>	
<i>localtime</i>																	
<i>object</i>																	

- <> - can be cast explicitly
- := - assignment cast is supported
- impl - implicit cast is supported
- \* - When casting a float type to an integer type, the fractional value naturally cannot be preserved after the cast. When executing this cast, we round to the nearest integer, rounding ties to the nearest even (e.g., 1.5 is rounded up to 2; 2.5 is also rounded to 2).

#### 8.1.6 Function calls

EdgeDB provides a number of functions in the *standard library*. It is also possible for users to *define their own* functions.

The syntax for a function call is as follows:

```
<function_name> "(" [<argument> [, <argument>, ...]] ")"  
  
# where <argument> is:  
  
<expr> | <identifier> := <expr>
```

Here <function\_name> is a possibly qualified name of a function, and <argument> is an *expression* optionally prefixed with an argument name and the assignment operator (:=) for *named only* arguments.

For example, the following computes the length of a string 'foo':

```
db> select len('foo');
{3}
```

And here's an example of using a *named only* argument to provide a default value:

```
db> select array_get(['hello', 'world'], 10, default := 'n/a');
{'n/a'}
```

See also
<a href="#">Schema &gt; Functions</a>
<a href="#">SDL &gt; Functions</a>
<a href="#">DDL &gt; Functions</a>
<a href="#">Introspection &gt; Functions</a>
<a href="#">Cheatsheets &gt; Functions</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; User-Defined Functions</a>

## 8.1.7 Cardinality

The number of items in a set is known as its **cardinality**. A set with a cardinality of zero is referred to as an **empty set**. A set with a cardinality of one is known as a **singleton**.

### 8.1.7.1 Terminology

The term **cardinality** is used to refer to both the *exact* number of elements in a given set or a *range* of possible values. Internally, EdgeDB tracks 5 different cardinality ranges: **Empty** (zero elements), **One** (a singleton set), **AtMostOne** (zero or one elements), **AtLeastOne** (one or more elements), and **Many** (any number of elements).

EdgeDB uses this information to statically check queries for validity. For instance, when assigning to a **required multi** link, the value being assigned in question *must* have a cardinality of **One** or **AtLeastOne** (as empty sets are not permitted).

### 8.1.7.2 Functions and operators

It's often useful to think of EdgeDB functions/operators as either *element-wise* or *aggregate*. Element-wise operations are applied to *each item* in a set. Aggregate operations operate on sets *as a whole*.

---

**Note:** This is a simplification, but it's a useful mental model when getting started with EdgeDB.

---

### 8.1.7.2.1 Aggregate operations

An example of an aggregate function is `count()`. It returns the number of elements in a given set. Regardless of the size of the input set, the result is a singleton integer.

```
db> select count('hello');
{1}
db> select count({'this', 'is', 'a', 'set'});
{4}
db> select count(<str>{});
{0}
```

Another example is `array_agg()`, which converts a *set* of elements into a singleton array.

```
db> select array_agg({1, 2, 3});
{[1, 2, 3]}
```

### 8.1.7.2.2 Element-wise operations

By contrast, the `len()` function is element-wise; it computes the length of each string inside a set of strings; as such, it converts a set of `str` into an equally-sized set of `int64`.

```
db> select len('hello');
{5}
db> select len({'hello', 'world'});
{5, 5}
```

### 8.1.7.2.3 Cartesian products

In case of element-wise operations that accept multiple arguments, the operation is applied to a cartesian product of all the input sets.

```
db> select {'aaa', 'bbb'} ++ {'ccc', 'ddd'};
{'aaaccc', 'aaadd', 'bbbccc', 'bbbdd'}
db> select {true, false} or {true, false};
{true, true, true, false}
```

By extension, if any of the input sets are empty, the result of applying an element-wise function is also empty. In effect, when EdgeDB detects an empty set, it “short-circuits” and returns an empty set without applying the operation.

```
db> select {} ++ {'ccc', 'ddd'};
{}
db> select {} or {true, false};
{}
```

---

**Note:** Certain functions and operators avoid this “short-circuit” behavior by marking their inputs as *optional*. A notable example of an operator with optional inputs is the `??` operator.

```
db> select <str>{} ?? 'default';
{'default'}
```

#### 8.1.7.2.4 Per-input cardinality

Ultimately, the distinction between “aggregate vs element-wise” operations is a false one. Consider the `in` operation.

```
db> select {1, 4} in {1, 2, 3};
{true, false}
```

This operator takes two inputs. If it was “element-wise” we would expect the cardinality of the above operation to the cartesian product of the input cardinalities:  $2 \times 3 = 6$ . If it was aggregate, we’d expect a singleton output.

Instead, the cardinality is 2. This operator is element-wise with respect to the first input and aggregate with respect to the second. The “element-wise vs aggregate” concept isn’t determined on a per-function/per-operator basis; it determined on a per-input basis.

#### 8.1.7.2.5 Type qualifiers

When defining functions, all inputs are element-wise by default. The `set` of *type qualifier* is used to designate an input as *aggregate*. Currently this modifier is not supported for user-defined functions, but it is used by certain standard library functions.

Similarly the `optional` qualifier marks the input as optional; an operation will be executed if an optional input is empty, whereas passing an empty set for a “standard” (non-optional) element-wise input will always result in an empty set.

Similarly, the *output* of a function *can be annotated* with `set` of and `optional` qualifiers.

#### 8.1.7.2.6 Cardinality computation

To compute the number of times a function/operator will be invoked, take the cardinality of each input and apply the following transformations, based on the type qualifier (or lack thereof) for each:

```
element-wise: N -> N
optional:      N -> max(1, N)
aggregate:     N -> 1
```

The ultimate cardinality of the result is the union of the results of each invocation; as such, it depends on the *values returned* by each invocation.

### 8.1.8 Select

**eql-statement**

**eql-haswith**

**index** order filter select offset limit with then asc desc first last empty

**select**—retrieve or compute a set of values.

```
[ with <with-item> [, ...] ]
select <expr>
[ filter <filter-expr> ]
```

(continues on next page)

(continued from previous page)

```
[ order by <order-expr> [direction] [then ...] ]
[ offset <offset-expr> ]
[ limit <limit-expr> ] ;
```

**filter** <filter-expr> The optional **filter** clause, where <filter-expr> is any expression that has a result of type `bool`. The condition is evaluated for every element in the set produced by the **select** clause. The result of the evaluation of the **filter** clause is a set of boolean values. If at least one value in this set is `true`, the input element is included, otherwise it is eliminated from the output.

**order by** <order-expr> [direction] [**then** ...] The optional **order by** clause has this general form:

```
order by
<order-expr> [ asc | desc ] [ empty { first | last } ]
[ then ... ]
```

The **order by** clause produces a result set sorted according to the specified expression or expressions, which are evaluated for every element of the input set.

If two elements are equal according to the leftmost *expression*, they are compared according to the next expression and so on. If two elements are equal according to all expressions, the resulting order is undefined.

Each *expression* can be an arbitrary expression that results in a value of an *orderable type*. Primitive types are orderable, object types are not. Additionally, the result of each expression must be an empty set or a singleton. Using an expression that may produce more elements is a compile-time error.

An optional `asc` or `desc` keyword can be added after any *expression*. If not specified `asc` is assumed by default.

If `empty last` is specified, then input values that produce an empty set when evaluating an *expression* are sorted *after* all other values; if `empty first` is specified, then they are sorted *before* all other values. If neither is specified, `empty first` is assumed when `asc` is specified or implied, and `empty last` when `desc` is specified.

**offset** <offset-expr> The optional **offset** clause, where <offset-expr> is a *singleton expression* of an integer type. This expression is evaluated once and its result is used to skip the first *element-count* elements of the input set while producing the output. If *element-count* evaluates to an empty set, it is equivalent to **offset 0**, which is equivalent to omitting the **offset** clause. If *element-count* evaluates to a value that is larger than the cardinality of the input set, an empty set is produced as the result.

**limit** <limit-expr> The optional **limit** clause, where <limit-expr> is a *singleton expression* of an integer type. This expression is evaluated once and its result is used to include only the first *element-count* elements of the input set while producing the output. If *element-count* evaluates to an empty set, it is equivalent to specifying no **limit** clause.

### 8.1.8.1 Description

**select** retrieves or computes a set of values. The data flow of a **select** block can be conceptualized like this:

```
with module example

# select clause
select
  <expr> # compute a set of things

# optional clause
```

(continues on next page)

(continued from previous page)

```
filter
    <expr> # filter the computed set

# optional clause
order by
    <expr> # define ordering of the filtered set

# optional clause
offset
    <expr> # slice the filtered/ordered set

# optional clause
limit
    <expr> # slice the filtered/ordered set
```

Please note that the `order by` clause defines ordering that can only be relied upon if the resulting set is not used in any other operation. `select`, `offset` and `limit` clauses are the only exception to that rule as they preserve the inherent ordering of the underlying set.

The first clause is `select`. It indicates that `filter`, `order by`, `offset`, or `limit` clauses may follow an expression, i.e. it makes an expression into a `select` statement. Without any of the optional clauses a `(select Expr)` is completely equivalent to `Expr` for any expression `Expr`.

Consider an example using the `filter` optional clause:

```
with module example
select User {
    name,
    owned := (select
        User.<owner[is Issue] {
            number,
            body
        }
    )
}
filter User.name like 'Alice%';
```

The above example retrieves a single user with a specific name. The fact that there is only one such user is a detail that can be well-known and important to the creator of the database, but otherwise non-obvious. However, forcing the cardinality to be at most 1 by using the `limit` clause ensures that a set with a single object or `{}` is returned. This way any further code that relies on the result of this query can safely assume there's only one result available.

```
with module example
select User {
    name,
    owned := (select
        User.<owner[is Issue] {
            number,
            body
        }
    )
}
filter User.name like 'Alice%'
limit 1;
```

Next example makes use of `order by` and `limit` clauses:

```
with module example
select Issue {
    number,
    body,
    due_date
}
filter
    exists Issue.due_date
    and
    Issue.status.name = 'Open'
order by
    Issue.due_date
limit 3;
```

The above query retrieves the top 3 open Issues with the closest due date.

### 8.1.8.2 Filter

The `filter` clause cannot affect anything aggregate-like in the preceding `select` clause. This is due to how `filter` clause works. It can be conceptualized as a function like `filter($input, set of $cond)`, where the `$input` represents the value of the preceding clause, while the `$cond` represents the filtering condition expression. Consider the following:

```
with module example
select count(User)
filter User.name like 'Alice%';
```

The above can be conceptualized as:

```
with module example
select _filter(
    count(User),
    User.name like 'Alice%'
);
```

In this form it is more apparent that `User` is a `set` of argument (of `count()`), while `User.name like 'Alice%`' is also a `set` of argument (of `filter`). So the symbol `User` in these two expressions exists in 2 parallel scopes. Contrast it with:

```
# This will actually only count users whose name starts with
# 'Alice'.

with module example
select count(
    (select User
        filter User.name like 'Alice%')
);

# which can be represented as:
with module example
select count(
```

(continues on next page)

(continued from previous page)

```
_filter(User,
        User.name like 'Alice%')
);
```

### 8.1.8.3 Clause signatures

Here is a summary of clauses that can be used with `select`:

- A filter `set of B`
- A `order by set of B`
- `set of A offset set of B`
- `set of A limit set of B`

See also
<a href="#">EdgeQL &gt; Select</a>
<a href="#">Cheatsheets &gt; Selecting data</a>

## 8.1.9 Insert

**eql-statement**

**eql-haswith**

`insert` – create a new object in a database

```
[ with <with-spec> [ , ... ] ]
insert <expression> [ <insert-shape> ]
[ unless conflict
  [ on <property-expr> [ else <alternative> ] ]
] ;
```

### 8.1.9.1 Description

`insert` inserts a new object into a database.

When evaluating an `insert` statement, *expression* is used solely to determine the *type* of the inserted object and is not evaluated in any other way.

If a value for a *required* link is evaluated to an empty set, an error is raised.

It is possible to insert multiple objects by putting the `insert` into a `for` statement.

See [Usage of for statement](#) for more details.

**with** Alias declarations.

The `with` clause allows specifying module aliases as well as expression aliases that can be referenced by the `update` statement. See [With block](#) for more information.

**<expression>** An arbitrary expression returning a set of objects to be updated.

```
insert <expression>
[ "{" <link> := <insert-value-expr> [, ...] "}" ]
```

**unless conflict [ on <property-expr> ]**

**index** unless conflict

Handler of conflicts.

This clause allows to handle specific conflicts arising during execution of `insert` without producing an error. If the conflict arises due to exclusive constraints on the properties specified by `property-expr`, then instead of failing with an error the `insert` statement produces an empty set (or an alternative result).

The exclusive constraint on `<property-expr>` cannot be defined on a parent type.

The specified `property-expr` may be either a reference to a property (or link) or a tuple of references to properties (or links). Although versions prior to 2.10 do *not* support `unless conflict` on *multi properties*, 2.10 adds support for these.

A caveat, however, is that `unless conflict` will not prevent conflicts caused between multiple DML operations in the same query; inserting two conflicting objects (through use of `for` or simply with two `insert` statements) will cause a constraint error.

Example:

```
insert User { email := 'user@example.org' }
unless conflict on .email
```

```
insert User { first := 'Jason', last := 'Momoa' }
unless conflict on (.first, .last)
```

**else <alternative>** Alternative result in case of conflict.

This clause can only appear after `unless conflict` clause. Any valid expression can be specified as the *alternative*. When a conflict arises, the result of the `insert` becomes the *alternative* expression (instead of the default `{}`).

In order to refer to the conflicting object in the *alternative* expression, the name used in the `insert` must be used (see [example below](#)).

### 8.1.9.2 Outputs

The result of an `insert` statement used as an *expression* is a singleton set containing the inserted object.

### 8.1.9.3 Examples

Here's a simple example of an `insert` statement creating a new user:

```
with module example
insert User {
    name := 'Bob Johnson'
};
```

`insert` is not only a statement, but also an expression and as such is has a value of the set of objects that has been created.

```
with module example
insert Issue {
    number := '100',
    body := 'Fix errors in insert',
    owner := (
        select User filter User.name = 'Bob Johnson'
    )
};
```

It is possible to create nested objects in a single `insert` statement as an atomic operation.

```
with module example
insert Issue {
    number := '101',
    body := 'Nested insert',
    owner := (
        insert User {
            name := 'Nested User'
        }
    )
};
```

The above statement will create a new `Issue` as well as a new `User` as the owner of the `Issue`. It will also return the new `Issue` linked to the new `User` if the statement is used as an expression.

It is also possible to create new objects based on some existing data either provided as an explicit list (possibly automatically generated by some tool) or a query. A `for` statement is the basis for this use-case and `insert` is simply the expression in the `union` clause.

```
# example of a bulk insert of users based on explicitly provided
# data
with module example
for x in {'Alice', 'Bob', 'Carol', 'Dave'}
union (insert User {
    name := x
});

# example of a bulk insert of issues based on a query
with
    module example,
    Elvis := (select User filter .name = 'Elvis'),
    Open := (select Status filter .name = 'Open')

for Q in (select User filter .name ilike 'A%')

union (insert Issue {
    name := Q.name + ' access problem',
    body := 'This user was affected by recent system glitch',
    owner := Elvis,
    status := Open
});
```

There's an important use-case where it is necessary to either insert a new object or update an existing one identified

with some key. This is what the `unless conflict` clause allows:

```
with module people
select (
    insert Person {
        name := "Łukasz Langa", is_admin := true
    }
    unless conflict on .name
    else (
        update Person
        set { is_admin := true }
    )
) {
    name,
    is_admin
};
```

---

**Note:** Statements in EdgeQL represent an atomic interaction with the database. From the point of view of a statement all side-effects (such as database updates) happen after the statement is executed. So as far as each statement is concerned, it is some purely functional expression evaluated on some specific input (database state).

---

See also
<a href="#">EdgeQL &gt; Insert</a>
<a href="#">Cheatsheets &gt; Inserting data</a>

## 8.1.10 Update

[eql-statement](#)

[eql-haswith](#)

update – update objects in a database

```
[ with <with-item> [, ...] ]

update <selector-expr>

[ filter <filter-expr> ]

set <shape> ;
```

update changes the values of the specified links in all objects selected by `update-selector-expr` and, optionally, filtered by `filter-expr`.

**with** Alias declarations.

The `with` clause allows specifying module aliases as well as expression aliases that can be referenced by the `update` statement. See [With block](#) for more information.

**update <selector-expr>** An arbitrary expression returning a set of objects to be updated.

**filter <filter-expr>** An expression of type `bool` used to filter the set of updated objects.

`<filter-expr>` is an expression that has a result of type `bool`. Only objects that satisfy the filter expression will be updated. See the description of the `filter` clause of the `select` statement for more information.

**set <shape>** A shape expression with the new values for the links of the updated object. There are three possible assignment operations permitted within the set shape:

```
set { <field> := <update-expr> [, ...] }
set { <field> += <update-expr> [, ...] }
set { <field> -= <update-expr> [, ...] }
```

The most basic assignment is the `:=`, which just sets the `<field>` to the specified `<update-expr>`. The `+=` and `-=` either add or remove the set of values specified by the `<update-expr>` from the *current* value of the `<field>`.

### 8.1.10.1 Output

On successful completion, an update statement returns the set of updated objects.

### 8.1.10.2 Examples

Here are a couple of examples of the update statement with simple assignments using `:=`:

```
# update the user with the name 'Alice Smith'
with module example
update User
filter .name = 'Alice Smith'
set {
    name := 'Alice J. Smith'
};

# update all users whose name is 'Bob'
with module example
update User
filter .name like 'Bob%'
set {
    name := User.name ++ '*'
};
```

For usage of `+=` and `-=` consider the following Post type:

```
# ... Assume some User type is already defined
type Post {
    required property title -> str;
    required property body -> str;
    # A "tags" property containing a set of strings
    multi property tags -> str;
    link author -> User;
}
```

```
# ... Assume some User type is already defined
type Post {
    required title: str;
    required body: str;
```

(continues on next page)

(continued from previous page)

```
# A "tags" property containing a set of strings
multi tags: str;
author: User;
}
```

The following queries add or remove tags from some user's posts:

```
with module example
update Post
filter .author.name = 'Alice Smith'
set {
    # add tags
    tags += {'example', 'edgeql'}
};

with module example
update Post
filter .author.name = 'Alice Smith'
set {
    # remove a tag, if it exist
    tags -= 'todo'
};
```

The statement `for <x> in <expr>` allows to express certain bulk updates more clearly. See [Usage of for statement](#) for more details.

See also
<a href="#">EdgeQL &gt; Update</a>
<a href="#">Cheatsheets &gt; Updating data</a>

### 8.1.11 Delete

[eql-statement](#)

[eql-haswith](#)

`delete` – remove objects from a database.

```
[ with <with-item> [, ...] ]

delete <expr>

[ filter <filter-expr> ]

[ order by <order-expr> [direction] [then ...] ]

[ offset <offset-expr> ]

[ limit <limit-expr> ] ;
```

**with** Alias declarations.

The `with` clause allows specifying module aliases as well as expression aliases that can be referenced by the `delete` statement. See [With block](#) for more information.

**delete ...** The entire `delete ...` statement is syntactic sugar for `delete (select ...)`. Therefore, the base `<expr>` and the following `filter`, `order by`, `offset`, and `limit` clauses shape the set to be deleted the same way an explicit `select` would.

### 8.1.11.1 Output

On successful completion, a `delete` statement returns the set of deleted objects.

### 8.1.11.2 Examples

Here's a simple example of deleting a specific user:

```
with module example
delete User
filter User.name = 'Alice Smith';
```

And here's the equivalent `delete (select ...)` statement:

```
with module example
delete (select User
        filter User.name = 'Alice Smith');
```

See also
<a href="#">EdgeQL &gt; Delete</a>
<a href="#">Cheatsheets &gt; Deleting data</a>

### 8.1.12 For

`eql-statement`

`eql-haswith`

`index` for union filter order offset limit

`for`-compute a union of subsets based on values of another set

```
[ with <with-item> [, ...] ]
for <variable> in <iterator-expr>
union <output-expr> ;
```

`for <variable> in <iterator-expr>` The `for` clause has this general form:

```
for <variable> in <iterator-expr>
```

`where <iterator-expr>` is a [literal](#), a [function call](#), a [set constructor](#), a [path](#), or any parenthesized expression or statement.

`union <output-expr>` The `union` clause of the `for` statement has this general form:

**union <output-expr>**

Here, `<output-expr>` is an arbitrary expression that is evaluated for every element in a set produced by evaluating the `for` clause. The results of the evaluation are appended to the result set.

**8.1.12.1 Usage of for statement**

`for` statement has some powerful features that deserve to be considered in detail separately. However, the common core is that `for` iterates over elements of some arbitrary expression. Then for each element of the iterator some set is computed and combined via a `union` with the other such computed sets.

The simplest use case is when the iterator is given by a set expression and it follows the general form of `for x in A ...:`

```
with module example
# the iterator is an explicit set of tuples, so x is an
# element of this set, i.e. a single tuple
for x in {
    (name := 'Alice', theme := 'fire'),
    (name := 'Bob', theme := 'rain'),
    (name := 'Carol', theme := 'clouds'),
    (name := 'Dave', theme := 'forest')
}
# typically this is used with an INSERT, DELETE or UPDATE
union (
    insert
        User {
            name := x.name,
            theme := x.theme,
        }
);
```

Since `x` is an element of a set it is guaranteed to be a non-empty singleton in all of the expressions used by the `union` and later clauses of `for`.

Another variation this usage of `for` is a bulk update. There are cases when a bulk update involves a lot of external data that cannot be derived from the objects being updated. That is a good use-case when a `for` statement is appropriate.

```
# Here's an example of an update that is awkward to
# express without the use of FOR statement
with module example
update User
filter .name in {'Alice', 'Bob', 'Carol', 'Dave'}
set {
    theme := 'red' if .name = 'Alice' else
        'star' if .name = 'Bob' else
            'dark' if .name = 'Carol' else
                'strawberry'
};

# Using a FOR statement, the above update becomes simpler to
# express or review for a human.
with module example
```

(continues on next page)

(continued from previous page)

```

for x in {
    (name := 'Alice', theme := 'red'),
    (name := 'Bob', theme := 'star'),
    (name := 'Carol', theme := 'dark'),
    (name := 'Dave', theme := 'strawberry')
}
union (
    update User
    filter .name = x.name
    set {
        theme := x.theme
    }
);

```

When updating data that mostly or completely depends on the objects being updated there's no need to use the `for` statement and it is not advised to use it for performance reasons.

```

with module example
update User
filter .name in {'Alice', 'Bob', 'Carol', 'Dave'}
set {
    theme := 'halloween'
};

# The above can be accomplished with a for statement,
# but it is not recommended.
with module example
for x in {'Alice', 'Bob', 'Carol', 'Dave'}
union (
    update User
    filter .name = x
    set {
        theme := 'halloween'
    }
);

```

Another example of using a `for` statement is working with link properties. Specifying the link properties either at creation time or in a later step with an update is often simpler with a `for` statement helping to associate the link target to the link property in an intuitive manner.

```

# Expressing this without for statement is fairly tedious.
with
    module example,
    U2 := User
for x in {
    (
        name := 'Alice',
        friends := [('Bob', 'coffee buff'),
                    ('Carol', 'dog person')]
    ),
    (
        name := 'Bob',

```

(continues on next page)

(continued from previous page)

```

    friends := [('Alice', 'movie buff'),
                ('Dave', 'cat person')]
)
}
union (
    update User
    filter .name = x.name
    set {
        friends := assert_distinct(
            (
                for f in array_unpack(x.friends)
                union (
                    select U2 {@nickname := f.1}
                    filter U2.name = f.0
                )
            )
        )
    }
);

```

See also
<a href="#">EdgeQL &gt; For</a>

### 8.1.13 Group

[eql-statement](#)

[eql-haswith](#)

[index](#) group using by

---

**Note:** The group statement is only available in EdgeDB 2.0 or later.

---

group—partition a set into subsets based on one or more keys

```
[ with <with-item> [, ...] ]

group [<alias> := ] <expr>

[ using <using-alias> := <expr>, [, ...] ]

by <grouping-element>, ... ;

# where a <grouping-element> is one of

<ref-or-list>
{ <grouping-element>, ... }
ROLLUP( <ref-or-list>, ... )
CUBE( <ref-or-list>, ... )
```

(continues on next page)

(continued from previous page)

```
# where a <ref-or-list> is one of
()
<grouping-ref>
( <grouping-ref>, ... )

# where a <grouping-ref> is one of
<using-alias>
.<field-name>
```

**group <expr>** The group clause sets up the input set that will be operated on.

Much like in `select` it is possible to define an ad-hoc alias at this stage to make referring to the starting set concisely.

**using <using-alias> := <expr>** The using clause defines one or more aliases which can then be used as part of the grouping key.

If the by clause only refers to `.<field-name>` the using clause is optional.

**by <grouping-element>** The by clause specifies which parameters will be used to partition the starting set.

There are only two basic components for defining `<grouping-element>`: references to `<using-alias>` defined in the using clause or by references to the short-path format of `.<field-name>`. The `.<field-name>` has to refer to properties or links immediately present on the type of starting set.

The basic building blocks can also be combined by using parentheses `( )` to indicate that partitioning will happen based on several parameters at once.

It is also possible to specify *grouping sets*, which are denoted using curly braces `{ }`. The results will contain different partitioning based on each of the grouping set elements. When there are multiple top-level grouping-elements then the cartesian product of them is taken to determine the grouping set. Thus `a`, `{b, c}` is equivalent to `{(a, b), (a, c)}` grouping sets.

ROLLUP and CUBE are a shorthand to specify particular grouping sets. ROLLUP groups by all prefixes of a list of elements, so ROLLUP `(a, b, c)` is equivalent to `{(), (a), (a, b), (a, b, c)}`. CUBE groups by all elements of the power set, so CUBE `(a, b)` is equivalent to `{(), (a), (b), (a, b)}`.

### 8.1.13.1 Output

The group statement partitions a starting set into subsets based on some specified parameters. The output is organized into a set of *free objects* of the following structure:

```
{
  "key": { <using-alias> := <value> [, ...] },
  "grouping": <set of keys used in grouping>,
  "elements": <the subset matching to the key>,
}
```

**"key"** The "key" contains another *free object*, which contains all the aliases or field names used as the key together with the specific values these parameters take for this particular subset.

**"grouping"** The "grouping" contains a `str` set of all the names of the parameters used as the key for this particular subset. This is especially useful when using grouping sets and the parameters used in the key are not the same for all partitionings.

"elements" The "elements" contains the actual subset of values that match the "key".

### 8.1.13.2 Examples

Here's a simple example without using any aggregation or any further processing:

```
db> group Movie {title} by .release_year;
{
  {
    key: {release_year: 2016},
    grouping: {'release_year'},
    elements: {
      default::Movie {title: 'Captain America: Civil War'},
      default::Movie {title: 'Doctor Strange'},
    },
  },
  {
    key: {release_year: 2017},
    grouping: {'release_year'},
    elements: {
      default::Movie {title: 'Spider-Man: Homecoming'},
      default::Movie {title: 'Thor: Ragnarok'},
    },
  },
  {
    key: {release_year: 2018},
    grouping: {'release_year'},
    elements: {default::Movie {title: 'Ant-Man and the Wasp'}},
  },
  {
    key: {release_year: 2019},
    grouping: {'release_year'},
    elements: {default::Movie {title: 'Spider-Man: No Way Home'}},
  },
  {
    key: {release_year: 2021},
    grouping: {'release_year'},
    elements: {default::Movie {title: 'Black Widow'}},
  },
  ...
}
```

Or we can group by an expression instead, such as whether the title starts with a vowel or not:

```
db> with
...   # Apply the group query only to more recent movies
...   M := (select Movie filter .release_year > 2015)
...   group M {title}
...   using vowel := re_test('(?i)^[aeiou]', .title)
...   by vowel;
{
  {
    key: {vowel: false},
```

(continues on next page)

(continued from previous page)

```

grouping: {'vowel'},
elements: {
    default::Movie {title: 'Thor: Ragnarok'},
    default::Movie {title: 'Doctor Strange'},
    default::Movie {title: 'Spider-Man: Homecoming'},
    default::Movie {title: 'Captain America: Civil War'},
    default::Movie {title: 'Black Widow'},
    default::Movie {title: 'Spider-Man: No Way Home'},
},
},
{
    key: {vowel: true},
    grouping: {'vowel'},
    elements: {default::Movie {title: 'Ant-Man and the Wasp'}},
},
}

```

It is also possible to group scalars instead of objects, in which case you need to define an ad-hoc alias to refer to the scalar set in order to specify how it will be grouped:

```

db> with
...     # Apply the group query only to more recent movies
...     M := (select Movie filter .release_year > 2015)
...     group T := M.title
...     using vowel := re_test('(?i)^[aeiou]', T)
...     by vowel;
{
{
    key: {vowel: false},
    grouping: {'vowel'},
    elements: {
        'Captain America: Civil War',
        'Doctor Strange',
        'Spider-Man: Homecoming',
        'Thor: Ragnarok',
        'Spider-Man: No Way Home',
        'Black Widow',
    },
},
{
    key: {vowel: true},
    grouping: {'vowel'},
    elements: {'Ant-Man and the Wasp'}
},
}

```

Often the results of `group` are immediately used in a `select` statement to provide some kind of analytical results:

```

db> with
...     # Apply the group query only to more recent movies
...     M := (select Movie filter .release_year > 2015),
...     groups := (

```

(continues on next page)

(continued from previous page)

```

...
  group M {title}
...
  using vowel := re_test('(?i)^[aeiou]', .title)
...
  by vowel
...
)
...
select groups {
...
  starts_with_vowel := .key.vowel,
...
  count := count(.elements),
...
  mean_title_length :=
    round(math::mean(len(.elements.title)))
...
};

{
  {starts_with_vowel: false, count: 6, mean_title_length: 18},
  {starts_with_vowel: true, count: 1, mean_title_length: 20},
}

```

It's possible to group by more than one parameter. For example, we can add the release decade to whether the title starts with a vowel:

```

db> with
...
  # Apply the group query only to more recent movies
...
  M := (select Movie filter .release_year > 2015),
...
  groups := (
...
    group M {title}
...
    using
      vowel := re_test('(?i)^[aeiou]', .title),
      decade := .release_year // 10
...
    by vowel, decade
...
)
...
select groups {
...
  key := .key {vowel, decade},
...
  count := count(.elements),
...
  mean_title_length :=
    math::mean(len(.elements.title))
...
};
{
  {
    key: {vowel: false, decade: 201},
    count: 5,
    mean_title_length: 19.8,
  },
  {
    key: {vowel: false, decade: 202},
    count: 1,
    mean_title_length: 11,
  },
  {
    key: {vowel: true, decade: 201},
    count: 1,
    mean_title_length: 20
  },
}

```

Having more than one grouping parameter opens up the possibility to using *grouping sets* to see the way grouping

parameters interact with the analytics we're gathering:

```
db> with
...    # Apply the group query only to more recent movies
...    M := (select Movie filter .release_year > 2015),
...    groups := (
...        group M {title}
...        using
...            vowel := re_test('(?i)^[aeiou]', .title),
...            decade := .release_year // 10
...        by CUBE(vowel, decade)
...    )
... select groups {
...     key := .key {vowel, decade},
...     grouping,
...     count := count(.elements),
...     mean_title_length :=
...         (math::mean(len(.elements.title)))
... } order by array_agg(.grouping);
{
{
    key: {vowel: {}, decade: {}},
    grouping: {},
    count: 7,
    mean_title_length: 18.571428571428573,
},
{
    key: {vowel: {}, decade: 202},
    grouping: {'decade'},
    count: 1,
    mean_title_length: 11,
},
{
    key: {vowel: {}, decade: 201},
    grouping: {'decade'},
    count: 6,
    mean_title_length: 19.833333333333332,
},
{
    key: {vowel: true, decade: {}},
    grouping: {'vowel'},
    count: 1,
    mean_title_length: 20,
},
{
    key: {vowel: false, decade: {}},
    grouping: {'vowel'},
    count: 6,
    mean_title_length: 18.333333333333332,
},
{
    key: {vowel: false, decade: 201},
    grouping: {'vowel', 'decade'},
```

(continues on next page)

(continued from previous page)

```

    count: 5,
    mean_title_length: 19.8,
},
{
    key: {vowel: true, decade: 201},
    grouping: {'vowel', 'decade'},
    count: 1,
    mean_title_length: 20,
},
{
    key: {vowel: false, decade: 202},
    grouping: {'vowel', 'decade'},
    count: 1,
    mean_title_length: 11,
},
}

```

See also
<a href="#">EdgeQL &gt; Group</a>

## 8.1.14 With block

**index** alias module

### keyword with

The `with` block in EdgeQL is used to define aliases.

The expression aliases are evaluated in the lexical scope they appear in, not the scope where their alias is used. This means that refactoring queries using aliases must be done with care so as not to alter the query semantics.

### 8.1.14.1 Specifying a module

#### keyword module

Used inside a `with` block to specify module names.

One of the more basic and common uses of the `with` block is to specify the default module that is used in a query. `with module <name>` construct indicates that whenever an identifier is used without any module specified explicitly, the module will default to `<name>` and then fall back to built-ins from `std` module.

The following queries are exactly equivalent:

```

with module example
select User {
    name,
    owned := (select
        User.<owner[is Issue] {
            number,
            body
        }
    )
}

```

(continues on next page)

(continued from previous page)

```
filter User.name like 'Alice%';

select example::User {
    name,
    owned := (select
        example::User.<owner[is example::Issue] {
            number,
            body
        }
    )
}
filter example::User.name like 'Alice%';
```

It is also possible to define aliased modules in the `with` block. Consider the following query that needs to compare objects corresponding to concepts defined in two different modules.

```
with
    module example,
    f as module foo
select User {
    name
}
filter .name = f::Foo.name;
```

Another use case is for giving short aliases to long module names (especially if module names contain `.`).

```
with
    module example,
    fbz as module foo.bar.baz
select User {
    name
}
filter .name = fbz::Baz.name;
```

### 8.1.14.2 Local Expression Aliases

It is possible to define an alias for an arbitrary expression. The result set of an alias expression behaves as a completely independent set of a given name. The contents of the set are determined by the expression at the point where the alias is defined. In terms of scope, the alias expression in the `with` block is in a sibling scope to the rest of the query.

It may be useful to factor out a common sub-expression from a larger complex query. This can be done by assigning the sub-expression a new symbol in the `with` block. However, care must be taken to ensure that this refactoring doesn't alter the meaning of the expression due to scope change.

All expression aliases defined in a `with` block must be referenced in the body of the query.

```
# Consider a query to get all users that own Issues and the
# comments those users made.
with module example
select Issue.owner {
    name,
    comments := Issue.owner.<owner[is Comment]
```

(continues on next page)

(continued from previous page)

```
};

# The above query can be refactored like this:
with
    module example,
    U := Issue.owner
select U {
    name,
    comments := U.<owner[is Comment]
};
```

An example of incorrect refactoring would be:

```
# This query gets a set of tuples of
# issues and their owners.
with
    module example
select (Issue, Issue.owner);

# This query gets a set of tuples that
# result from a cartesian product of all issues
# with all owners. This is because ``Issue`` and ``U``
# are considered independent sets.
with
    module example,
    U := Issue.owner
select (Issue, U);
```

### 8.1.14.3 Detached

#### keyword detached

The detached keyword marks an expression as not belonging to any scope.

A detached expression allows referring to some set as if it were defined in the top-level `with` block. Basically, detached expressions ignore all current scopes they are nested in and only take into account module aliases. The net effect is that it is possible to refer to an otherwise related set as if it were unrelated:

```
with module example
update User
filter .name = 'Dave'
set {
    friends := (select detached User filter .name = 'Alice'),
    coworkers := (select detached User filter .name = 'Bob')
};
```

Here you can use the `detached User` expression, rather than having to define `U := User` in the `with` block just to allow it to be used in the body of the `update`. The goal is to indicate that the `User` in the `update` body is not in any way related to the `User` that's being updated.

See also
<a href="#">EdgeQL &gt; With</a>

## 8.1.15 Start transaction

### eql-statement

`start transaction` – start a transaction

```
start transaction <transaction-mode> [ , ... ] ;  
  
# where <transaction-mode> is one of:  
  
isolation serializable  
read write | read only  
deferrable | not deferrable
```

### 8.1.15.1 Description

This command starts a new transaction block.

Any EdgeDB command outside of an explicit transaction block starts an implicit transaction block; the transaction is then automatically committed if the command was executed successfully, or automatically rolled back if there was an error. This behavior is often called “autocommit”.

### 8.1.15.2 Parameters

The `<transaction-mode>` can be one of the following:

**isolation serializable** All statements of the current transaction can only see data changes committed before the first query or data-modification statement was executed in this transaction. If a pattern of reads and writes among concurrent serializable transactions would create a situation which could not have occurred for any serial (one-at-a-time) execution of those transactions, one of them will be rolled back with a `serialization_failure` error.

**read write** Sets the transaction access mode to read/write.

This is the default.

**read only** Sets the transaction access mode to read-only. Any data modifications with `insert`, `update`, or `delete` are disallowed. Schema mutations via `DDL` are also disallowed.

**deferrable** The transaction can be set to deferrable mode only when it is `serializable` and `read only`. When all three of these properties are selected for a transaction, the transaction may block when first acquiring its snapshot, after which it is able to run without the normal overhead of a `serializable` transaction and without any risk of contributing to or being canceled by a serialization failure. This mode is well suited for long-running reports or backups.

### 8.1.15.3 Examples

Start a new transaction and rollback it:

```
start transaction;  
select 'Hello World!';  
rollback;
```

Start a serializable deferrable transaction:

```
start transaction isolation serializable, read only, deferrable;
```

See also
<a href="#">Reference &gt; EdgeQL &gt; Commit</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback</a>
<a href="#">Reference &gt; EdgeQL &gt; Declare savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback to savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Release savepoint</a>

## 8.1.16 Commit

### eql-statement

`commit` – commit the current transaction

```
commit ;
```

### 8.1.16.1 Example

Commit the current transaction:

```
commit;
```

### 8.1.16.2 Description

The `commit` command commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

See also
<a href="#">Reference &gt; EdgeQL &gt; Start transaction</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback</a>
<a href="#">Reference &gt; EdgeQL &gt; Declare savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback to savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Release savepoint</a>

## 8.1.17 Rollback

### eql-statement

`rollback` – abort the current transaction

```
rollback ;
```

### 8.1.17.1 Example

Abort the current transaction:

```
rollback;
```

### 8.1.17.2 Description

The `rollback` command rolls back the current transaction and causes all updates made by the transaction to be discarded.

See also
<a href="#">Reference &gt; EdgeQL &gt; Start transaction</a>
<a href="#">Reference &gt; EdgeQL &gt; Commit</a>
<a href="#">Reference &gt; EdgeQL &gt; Declare savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback to savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Release savepoint</a>

### 8.1.18 Declare savepoint

#### eql-statement

`declare savepoint` – declare a savepoint within the current transaction

```
declare savepoint <savepoint-name> ;
```

### 8.1.18.1 Description

`savepoint` establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

It is an error to declare a savepoint outside of a transaction.

### 8.1.18.2 Example

```
# Will select no objects:
select test::TestObject { name };

start transaction;

    insert test::TestObject { name := 'q1' };
    insert test::TestObject { name := 'q2' };

    # Will select two TestObjects with names 'q1' and 'q2'
    select test::TestObject { name };

    declare savepoint f1;
        insert test::TestObject { name:='w1' };
```

(continues on next page)

(continued from previous page)

```

# Will select three TestObjects with names
# 'q1' 'q2', and 'w1'
select test::TestObject { name };
rollback to savepoint f1;

# Will select two TestObjects with names 'q1' and 'q2'
select test::TestObject { name };

rollback;

```

**See also**

- [Reference > EdgeQL > Start transaction](#)
- [Reference > EdgeQL > Commit](#)
- [Reference > EdgeQL > Rollback](#)
- [Reference > EdgeQL > Rollback to savepoint](#)
- [Reference > EdgeQL > Release savepoint](#)

## 8.1.19 Release savepoint

### eql-statement

`release savepoint` – release a previously declared savepoint

```
release savepoint <savepoint-name> ;
```

### 8.1.19.1 Description

`release savepoint` destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see [rollback to savepoint](#).)

`release savepoint` also destroys all savepoints that were established after the named savepoint was established.

### 8.1.19.2 Example

```

start transaction;
# ...
declare savepoint f1;
# ...
release savepoint f1;
# ...
rollback;

```

See also
<a href="#">Reference &gt; EdgeQL &gt; Start transaction</a>
<a href="#">Reference &gt; EdgeQL &gt; Commit</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback</a>
<a href="#">Reference &gt; EdgeQL &gt; Declare savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback to savepoint</a>

## 8.1.20 Rollback to savepoint

### eql-statement

rollback to savepoint – rollback to a savepoint within the current transaction

```
rollback to savepoint <savepoint-name> ;
```

### 8.1.20.1 Description

Rollback all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

rollback to savepoint implicitly destroys all savepoints that were established after the named savepoint.

### 8.1.20.2 Example

```
start transaction;
# ...
declare savepoint f1;
# ...
rollback to savepoint f1;
# ...
rollback;
```

See also
<a href="#">Reference &gt; EdgeQL &gt; Start transaction</a>
<a href="#">Reference &gt; EdgeQL &gt; Commit</a>
<a href="#">Reference &gt; EdgeQL &gt; Rollback</a>
<a href="#">Reference &gt; EdgeQL &gt; Declare savepoint</a>
<a href="#">Reference &gt; EdgeQL &gt; Release savepoint</a>

## 8.1.21 Set

### eql-statement

set – set one or multiple session-level parameters

```
set module <module> ;
set alias <alias> as module <module> ;
set global <name> := <expr> ;
```

### 8.1.21.1 Description

This command allows altering the configuration of the current session.

### 8.1.21.2 Variations

**set module <module>** Set the default module for the current section to *module*.

For example, if a module `foo` contains type `FooType`, the following is how the type can be referred to:

```
# Use the fully-qualified name.  
select foo::FooType;  
  
# Use the WITH clause to define the default module  
# for the query.  
with module foo select foo::FooType;  
  
# Set the default module for the current session ...  
set module foo;  
# ... and use an unqualified name.  
select FooType;
```

**set alias <alias> as module <module>** Define <alias> for the <module>.

For example:

```
# Use the fully-qualified name.  
select foo::FooType;  
  
# Use the WITH clause to define a custom alias  
# for the "foo" module.  
with bar as module foo  
select bar::FooType;  
  
# Define "bar" as an alias for the "foo" module for  
# the current session ...  
set alias bar as module foo;  
# ... and use "bar" instead of "foo".  
select bar::FooType;
```

**set global <name> := <expr>** Set the global variable *name* to the specified value.

For example:

```
# Set the global variable "current_user_id".  
set global current_user_id :=  
    <uuid>'00ea8eaa-02f9-11ed-a676-6bd11cc6c557';  
  
# We can now use that value in a query.  
select User { name }  
filter .id = global current_user_id;
```

### 8.1.21.3 Examples

```
set module foo;

set alias foo AS module std;

set global current_user_id :=
<uuid>'00ea8eaa-02f9-11ed-a676-6bd11cc6c557';
```

**See also**

[Reference > EdgeQL > Reset](#)

## 8.1.22 Reset

### eql-statement

**reset** – reset one or multiple session-level parameters

```
reset module ;
reset alias <alias> ;
reset alias * ;
reset global <name> ;
```

### 8.1.22.1 Description

This command allows resetting one or many configuration parameters of the current session.

### 8.1.22.2 Variations

**reset module** Reset the default module name back to “default” for the current session.

For example, if a module `foo` contains type `FooType`, the following is how the `set` and `reset` commands can be used to alias it:

```
# Set the default module to "foo" for the current session.
set module foo;

# This query is now equivalent to "select foo::FooType".
select FooType;

# Reset the default module for the current session.
reset module;

# This query will now produce an error.
select FooType;
```

**reset alias <alias>** Reset `<alias>` for the current session.

For example:

```
# Alias the "std" module as "foo".
set alias foo as module std;

# Now "std::min()" can be called as "foo::min()" in
# the current session.
select foo::min({1});

# Reset the alias.
reset alias foo;

# Now this query will error out, as there is no
# module "foo".
select foo::min({1});
```

**reset alias \*** Reset all aliases defined in the current session. This command affects aliases set with `set alias` and `set module`. The default module will be set to “default”.

Example:

```
# Reset all custom aliases for the current session.
reset alias *;
```

**reset global <name>** Reset the global variable *name* to its default value or {} if the variable has no default value and is optional.

### 8.1.22.3 Examples

```
reset module;

reset alias foo;

reset alias *;

reset global current_user_id;
```

<b>See also</b>
<a href="#">Reference &gt; EdgeQL &gt; Set</a>

### 8.1.23 Describe

#### eql-statement

`describe` – provide human-readable description of a schema or a schema object

```
describe schema [ as {ddl | sdl | test [ verbose ]} ];
describe <schema-type> <name> [ as {ddl | sdl | text [ verbose ]} ];
# where <schema-type> is one of
object
```

(continues on next page)

(continued from previous page)

<b>annotation</b>
<b>constraint</b>
<b>function</b>
<b>link</b>
<b>module</b>
<b>property</b>
<b>scalar type</b>
<b>type</b>

### 8.1.23.1 Description

`describe` generates a human-readable description of a schema object.

The output of a `describe` command is a `str`, although it cannot be used as an expression in queries.

There are three output formats to choose from:

**as `ddl`** Provide a valid `DDL` definition.

The `DDL` generated is a complete valid definition of the particular schema object assuming all the other referenced schema objects already exist.

This is the default format.

**as `sdl`** Provide an `SDL` definition.

The `SDL` generated is a complete valid definition of the particular schema object assuming all the other referenced schema objects already exist.

**as `text [verbose]`** Provide a human-oriented definition.

The human-oriented definition generated is similar to `SDL`, but it includes all the details that are inherited (if any).

The `verbose` mode enables displaying additional details, such as `annotations` and `constraints`, which are otherwise omitted.

When the `describe` command is used with the `schema` the result is a definition of the entire database schema. Only the `as ddl` option is available for schema description.

The `describe` command can specify the type of schema object that it should generate the description of:

**object <name>** Match any module level schema object with the specified `name`.

This is the most general use of the `describe` command. It does not match `modules` (and other globals that cannot be uniquely identified just by the name).

**annotation <name>** Match only `annotations` with the specified `name`.

**constraint <name>** Match only `constraints` with the specified `name`.

**function <name>** Match only `functions` with the specified `name`.

**link <name>** Match only `links` with the specified `name`.

**module <name>** Match only `modules` with the specified `name`.

**property <name>** Match only `properties` with the specified `name`.

**scalar type <name>** Match only `scalar types` with the specified `name`.

**type <name>** Match only `object types` with the specified `name`.

### 8.1.23.2 Examples

Consider the following schema:

```
abstract type Named {
    required property name -> str {
        delegated constraint exclusive;
    }
}

type User extending Named {
    required property email -> str {
        annotation title := 'Contact email';
    }
}
```

```
abstract type Named {
    required name: str {
        delegated constraint exclusive;
    }
}

type User extending Named {
    required email: str {
        annotation title := 'Contact email';
    }
}
```

Here are some examples of a `describe` command:

```
db> describe object User;
{
    "create type default::User extending default::Named {
        create required single property email -> std::str {
            create annotation std::title := 'Contact email';
        };
    };"}
}
db> describe object User as sdl;
{
    "type default::User extending default::Named {
        required single property email -> std::str {
            annotation std::title := 'Contact email';
        };
    };"}
}
db> describe object User as text;
{
    'type default::User extending default::Named {
        required single link __type__ -> schema::Type {
            readonly := true;
        };
        required single property email -> std::str;
```

(continues on next page)

(continued from previous page)

```

required single property id -> std::uuid {
    readonly := true;
};
    required single property name -> std::str;
};"}
}
db> describe object User as text verbose;
{
    "type default::User extending default::Named {
        required single link __type__ -> schema::Type {
            readonly := true;
        };
        required single property email -> std::str {
            annotation std::title := 'Contact email';
        };
        required single property id -> std::uuid {
            readonly := true;
            constraint std::exclusive;
        };
        required single property name -> std::str {
            constraint std::exclusive;
        };
    };
};"}
}
db> describe schema;
{
    "create module default if not exists;
create abstract type default::Named {
    create required single property name -> std::str {
        create delegated constraint std::exclusive;
    };
};
create type default::User extending default::Named {
    create required single property email -> std::str {
        create annotation std::title := 'Contact email';
    };
};"}
}

```

The `describe` command also warns you if there are standard library matches that are masked by some user-defined object. Consider the following schema:

```

module default {
    function len(v: tuple<float64, float64>) -> float64 using (
        select (v.0 ^ 2 + v.1 ^ 2) ^ 0.5
    );
}

```

So within the `default` module the user-defined function `len` (computing the length of a vector) masks the built-ins:

```

db> describe function len as text;
{

```

(continues on next page)

(continued from previous page)

```

'function default::len(v: tuple<std::float64, std::float64>) ->
std::float64 using (select
  (((v.0 ^ 2) + (v.1 ^ 2)) ^ 0.5)
);

# The following builtins are masked by the above:

# function std::len(array: array<anytype>) -> std::int64 {
#   volatility := \Immutable\';
#   annotation std::description := \A polymorphic function to calculate
a "length" of its first argument.\';
#   using sql $$ 
#   SELECT cardinality("array")::bigint
#   $$ 
# };
# function std::len(bytes: std::bytes) -> std::int64 {
#   volatility := \Immutable\';
#   annotation std::description := \A polymorphic function to calculate
a "length" of its first argument.\';
#   using sql $$ 
#   SELECT length("bytes")::bigint
#   $$ 
# };
# function std::len(str: std::str) -> std::int64 {
#   volatility := \Immutable\';
#   annotation std::description := \A polymorphic function to calculate
a "length" of its first argument.\';
#   using sql $$ 
#   SELECT char_length("str")::bigint
#   $$ 
# };',
}

```

## 8.2 SDL

### **edb-alt-title** Schema Definition Language

This section describes the high-level language used to define EdgeDB schema. It is called the EdgeDB *schema definition language* or *SDL*. There's a correspondence between this declarative high-level language and the imperative low-level *DDL*.

SDL is a declarative language optimized for human readability and expressing the state of the EdgeDB schema without getting into the details of how to arrive at that state. Each *SDL* block represents the complete schema state for a given *database*.

Syntactically, an SDL declaration mirrors the `create` DDL for the corresponding entity, but with all of the `create` and `set` keywords omitted. The typical SDL structure is to use *module blocks* with the rest of the declarations being nested in their respective modules.

EdgeDB 3.0 introduces a new SDL syntax which diverges slightly from DDL. The old SDL syntax is still fully supported, but the new syntax allows for cleaner and less verbose expression of your schemas.

- Pointers no longer require an arrow (`->`). You may instead use a colon after the name of the link or property.

- The `link` and `property` keywords are now optional for non-computed pointers when the target type is explicitly specified.

That means that this type definition:

```
type User {
    required property email -> str;
}
```

could be replaced with this equivalent one in EdgeDB 3+:

```
type User {
    required email: str;
}
```

When reading our documentation, the version selection dropdown will update the syntax of most SDL examples to the preferred syntax for the version selected. This is only true for versioned sections of the documentation.

Since SDL is declarative in nature, the specific order of declarations of module blocks or individual items does not matter.

The built-in *migration tools* expect the schema to be given in SDL format. For example:

```
# "default" module block
module default {
    type Movie {
        required property title -> str;
        # the year of release
        property year -> int64;
        required link director -> Person;
        required multi link actors -> Person;
    }
    type Person {
        required property first_name -> str;
        required property last_name -> str;
    }
}
```

```
# "default" module block
module default {
    type Movie {
        required title: str;
        # the year of release
        year: int64;
        required director: Person;
        required multi actors: Person;
    }
    type Person {
        required first_name: str;
        required last_name: str;
    }
}
```

It is possible to also omit the module blocks, but then individual declarations must use *fully-qualified names* so that they can be assigned to their respective modules. For example, the following is equivalent to the previous migration:

```
# no module block
type default::Movie {
    required property title -> str;
    # the year of release
    property year -> int64;
    required link director -> default::Person;
    required multi link actors -> default::Person;
}
type default::Person {
    required property first_name -> str;
    required property last_name -> str;
}
```

```
# no module block
type default::Movie {
    required title: str;
    # the year of release
    year: int64;
    required director: default::Person;
    required multi actors: default::Person;
}
type default::Person {
    required first_name: str;
    required last_name: str;
}
```

## 8.2.1 Modules

This section describes the SDL commands pertaining to *modules*.

### 8.2.1.1 Example

Declare an empty module:

```
module my_module {}
```

Declare a module with some content:

```
module my_module {
    type User {
        required property name -> str;
    }
}
```

```
module my_module {
    type User {
        required name: str;
    }
}
```

### 8.2.1.2 Syntax

Define a module corresponding to the [more explicit DDL commands](#).

```
module <ModuleName> "{"
    [ <schema-declarations> ]
    ...
}"
```

Define a nested module.

```
module <ParentModuleName> "{"
    [ <schema-declarations> ]
    module <ModuleName> "{"
        [ <schema-declarations> ]
    }"
    ...
}"
```

### 8.2.1.3 Description

The module block declaration defines a new module similar to the `create module` command, but it also allows putting the module content as nested declarations:

**<schema-declarations>** Define various schema items that belong to this module.

Unlike `create module` command, a module block with the same name can appear multiple times in an SDL document. In that case all blocks with the same name are merged into a single module under that name. For example:

```
module my_module {
    abstract type Named {
        required property name -> str;
    }
}

module my_module {
    type User extending Named;
}
```

```
module my_module {
    abstract type Named {
        required name: str;
    }
}

module my_module {
    type User extending Named;
}
```

The above is equivalent to:

```
module my_module {
    abstract type Named {
```

(continues on next page)

(continued from previous page)

```

required property name -> str;
}

type User extending Named;
}

```

```

module my_module {
    abstract type Named {
        required name: str;
    }

    type User extending Named;
}

```

Typically, in the documentation examples of SDL the *module block* is omitted and instead its contents are described without assuming which specific module they belong to.

It's also possible to declare modules implicitly. In this style SDL declaration uses *fully-qualified name* for the item that is being declared. The *module* part of the *fully-qualified* name implies that a module by that name will be automatically created in the schema. The following declaration is equivalent to the previous examples, but it declares module `my_module` implicitly:

```

abstract type my_module::Named {
    required property name -> str;
}

type my_module::User extending my_module::Named;

```

```

abstract type my_module::Named {
    required name: str;
}

type my_module::User extending my_module::Named;

```

A module block can be nested inside another module block to create a nested module. If you want reference an entity in a nested module by its fully-qualified name, you will need to reference all of the containing modules' names: `<ParentModuleName>::<ModuleName>::<EntityName>`

## 8.2.2 Object Types

This section describes the SDL declarations pertaining to *object types*.

### 8.2.2.1 Example

Consider a User type with a few properties:

```
type User {
    # define some properties and a link
    required property name -> str;
    property address -> str;

    multi link friends -> User;

    # define an index for User based on name
    index on (__subject__.name);
}
```

```
type User {
    # define some properties and a link
    required name: str;
    address: str;

    multi friends: User;

    # define an index for User based on name
    index on (__subject__.name);
}
```

An alternative way to define the same User type could be by using abstract types. These abstract types can then be re-used in other type definitions as well:

```
abstract type Named {
    required property name -> str;
}

abstract type HasAddress {
    property address -> str;
}

type User extending Named, HasAddress {
    # define some user-specific properties and a link
    multi link friends -> User;

    # define an index for User based on name
    index on (__subject__.name);
}
```

```
abstract type Named {
    required name: str;
}
```

(continues on next page)

(continued from previous page)

```
abstract type HasAddress {
    address: str;
}

type User extending Named, HasAddress {
    # define some user-specific properties and a link
    multi friends: User;

    # define an index for User based on name
    index on (__subject__.name);
}
```

Introducing abstract types opens up the possibility of *polymorphic queries*.

### 8.2.2.2 Syntax

Define a new object type corresponding to the *more explicit DDL commands*.

```
[abstract] type <TypeName> [extending <supertype> [, ...] ]
[ "{"
    [ <annotation-declarations> ]
    [ <property-declarations> ]
    [ <link-declarations> ]
    [ <constraint-declarations> ]
    [ <index-declarations> ]
    ...
"}"]
```

### 8.2.2.3 Description

This declaration defines a new object type with the following options:

**abstract** If specified, the created type will be *abstract*.

<TypeName> The name (optionally module-qualified) of the new type.

**extending** <supertype> [, ...] Optional clause specifying the *supertypes* of the new type.

Use of **extending** creates a persistent type relationship between the new subtype and its supertype(s). Schema modifications to the supertype(s) propagate to the subtype.

References to supertypes in queries will also include objects of the subtype.

If the same *link* name exists in more than one supertype, or is explicitly defined in the subtype and at least one supertype, then the data types of the link targets must be *compatible*. If there is no conflict, the links are merged to form a single link in the new type.

These sub-declarations are allowed in the Type block:

<**annotation-declarations**> Set object type *annotation* to a given *value*.

<**property-declarations**> Define a concrete *property* for this object type.

<**link-declarations**> Define a concrete *link* for this object type.

**<constraint-declarations>** Define a concrete *constraint* for this object type.

**<index-declarations>** Define an *index* for this object type.

See also
<a href="#">Schema &gt; Object types</a>
<a href="#">DDL &gt; Object types</a>
<a href="#">Introspection &gt; Object types</a>
<a href="#">Cheatsheets &gt; Object types</a>

## 8.2.3 Scalar Types

This section describes the SDL declarations pertaining to *scalar types*.

### 8.2.3.1 Example

Declare a new non-negative integer type:

```
scalar type posint64 extending int64 {
    constraint min_value(0);
}
```

### 8.2.3.2 Syntax

Define a new scalar type corresponding to the *more explicit DDL commands*.

```
[abstract] scalar type <TypeName> [extending <supertype> [, ...] ]
[ "{"
    [ <annotation-declarations> ]
    [ <constraint-declarations> ]
    ...
"}" ]
```

### 8.2.3.3 Description

This declaration defines a new object type with the following options:

**abstract** If specified, the created scalar type will be *abstract*.

**<TypeName>** The name (optionally module-qualified) of the new scalar type.

**extending <supertype>** Optional clause specifying the *supertype* of the new type.

If <supertype> is an *enumerated type* declaration then an enumerated scalar type is defined.

Use of *extending* creates a persistent type relationship between the new subtype and its supertype(s). Schema modifications to the supertype(s) propagate to the subtype.

The valid SDL sub-declarations are listed below:

**<annotation-declarations>** Set scalar type *annotation* to a given *value*.

**<constraint-declarations>** Define a concrete *constraint* for this scalar type.

## 8.2.4 Links

This section describes the SDL declarations pertaining to *links*.

### 8.2.4.1 Examples

Declare an *abstract* link “friends\_base” with a helpful title:

```
abstract link friends_base {
    # declare a specific title for the link
    annotation title := 'Close contacts';
}
```

Declare a *concrete* link “friends” within a “User” type:

```
type User {
    required property name -> str;
    property address -> str;
    # define a concrete link "friends"
    multi link friends extending friends_base -> User;

    index on (__subject__.name);
}
```

```
type User {
    required name: str;
    address: str;
    # define a concrete link "friends"
    multi link friends: User {
        extending friends_base;
    };

    index on (__subject__.name);
}
```

#### 8.2.4.1.1 Overloading

Any time that the SDL declaration refers to an inherited link that is being overloaded (by adding more constraints or changing the target type, for example), the `overloaded` keyword must be used. This is to prevent unintentional overloading due to name clashes:

```
abstract type Friendly {
    # this type can have "friends"
    multi link friends -> Friendly;
}

type User extending Friendly {
    # overload the link target to be User, specifically
    overloaded multi link friends -> User;
    # ... other links and properties
}
```

```

abstract type Friendly {
    # this type can have "friends"
    multi friends: Friendly;
}

type User extending Friendly {
    # overload the link target to be User, specifically
    overloaded multi friends: User;
    # ... other links and properties
}

```

#### 8.2.4.2 Syntax

Define a new link corresponding to the *more explicit DDL commands*.

```

# Concrete link form used inside type declaration:
[ overloaded ] [{required | optional}] [{single | multi}]
  link <name>
    [ extending <base> [, ...] ] -> <type>
    [ "{"
      [ default := <expression> ; ]
      [ readonly := {true | false} ; ]
      [ on target delete <action> ; ]
      [ <annotation-declarations> ]
      [ <property-declarations> ]
      [ <constraint-declarations> ]
      ...
    "}
    ]
  }

# Computed link form used inside type declaration:
[{required | optional}] [{single | multi}]
  link <name> := <expression>;

# Computed link form used inside type declaration (extended):
[ overloaded ] [{required | optional}] [{single | multi}]
  link <name>
    [ extending <base> [, ...] ] [-> <type>]
    [ "{"
      using (<expression>) ;
      [ <annotation-declarations> ]
      [ <constraint-declarations> ]
      ...
    "}
    ]
  }

# Abstract link form:
abstract link <name> [extending <base> [, ...]]
[ "{"
  [ readonly := {true | false} ; ]
  [ <annotation-declarations> ]
  [ <property-declarations> ]
  [ <constraint-declarations> ]
}
```

(continues on next page)

(continued from previous page)

```
[ <index-declarations> ]
...
"}" ]
```

```
# Concrete link form used inside type declaration:
[ overloaded ] [{required | optional}] [{single | multi}]
[ link ] <name> : <type>
[ "{"
  [ extending <base> [, ...] ; ]
  [ default := <expression> ; ]
  [ readonly := {true | false} ; ]
  [ on target delete <action> ; ]
  [ <annotation-declarations> ]
  [ <property-declarations> ]
  [ <constraint-declarations> ]
  ...
"}" ]
```

```
# Computed link form used inside type declaration:
[{required | optional}] [{single | multi}]
link <name> := <expression>;
```

```
# Computed link form used inside type declaration (extended):
[ overloaded ] [{required | optional}] [{single | multi}]
link <name> [: <type>]
[ "{"
  using (<expression>) ;
  [ extending <base> [, ...] ; ]
  [ <annotation-declarations> ]
  [ <constraint-declarations> ]
  ...
"}" ]
```

```
# Abstract link form:
abstract link <name>
[ "{"
  [extending <base> [, ...] ; ]
  [ readonly := {true | false} ; ]
  [ <annotation-declarations> ]
  [ <property-declarations> ]
  [ <constraint-declarations> ]
  [ <index-declarations> ]
  ...
"}" ]
```

### 8.2.4.3 Description

There are several forms of link declaration, as shown in the syntax synopsis above. The first form is the canonical definition form, the second form is used for defining a *computed link*, and the last form is used to define an abstract link. The abstract form allows declaring the link directly inside a *module*. Concrete link forms are always used as sub-declarations of an *object type*.

The following options are available:

**overloaded** If specified, indicates that the link is inherited and that some feature of it may be altered in the current object type. It is an error to declare a link as *overloaded* if it is not inherited.

**required** If specified, the link is considered *required* for the parent object type. It is an error for an object to have a required link resolve to an empty value. Child links **always** inherit the *required* attribute, i.e. it is not possible to make a required link non-required by extending it.

**optional** This is the default qualifier assumed when no qualifier is specified, but it can also be specified explicitly. The link is considered *optional* for the parent object type, i.e. it is possible for the link to resolve to an empty value.

**multi** Specifies that there may be more than one instance of this link in an object, in other words, `Object.link` may resolve to a set of a size greater than one.

**single** Specifies that there may be at most *one* instance of this link in an object, in other words, `Object.link` may resolve to a set of a size not greater than one. **single** is assumed if neither **multi** nor **single** qualifier is specified.

**extending <base> [, ...]** Optional clause specifying the *parents* of the new link item.

Use of *extending* creates a persistent schema relationship between the new link and its parents. Schema modifications to the parent(s) propagate to the child.

If the same *property* name exists in more than one parent, or is explicitly defined in the new link and at least one parent, then the data types of the property targets must be *compatible*. If there is no conflict, the link properties are merged to form a single property in the new link item.

As of EdgeDB 3.0, the *extended* clause is now a sub-declaration of the link and included inside the curly braces rather than an option as in earlier versions.

**<type>** The type must be a valid *type expression* denoting an object type.

The valid SDL sub-declarations are listed below:

**default := <expression>** Specifies the default value for the link as an EdgeQL expression. The default value is used in an *insert* statement if an explicit value for this link is not specified.

**readonly := {true | false}** If **true**, the link is considered *read-only*. Modifications of this link are prohibited once an object is created. All of the derived links **must** preserve the original *read-only* value.

**<annotation-declarations>** Set link *annotation* to a given *value*.

**<property-declarations>** Define a concrete *property* on the link.

**<constraint-declarations>** Define a concrete *constraint* on the link.

**<index-declarations>** Define an *index* for this abstract link. Note that this index can only refer to link properties.

See also
<a href="#">Schema &gt; Links</a>
<a href="#">DDL &gt; Links</a>
<a href="#">Introspection &gt; Object types</a>

## 8.2.5 Properties

This section describes the SDL declarations pertaining to *properties*.

### 8.2.5.1 Examples

Declare an *abstract* property “address\_base” with a helpful title:

```
abstract property address_base {
    # declare a specific title for the link
    annotation title := 'Mailing address';
}
```

Declare *concrete* properties “name” and “address” within a “User” type:

```
type User {
    # define concrete properties
    required property name -> str;
    property address extending address_base -> str;

    multi link friends -> User;

    index on (__subject__.name);
}
```

```
type User {
    # define concrete properties
    required name: str;
    property address: str {
        extending address_base;
    };

    multi friends: User;

    index on (__subject__.name);
}
```

Any time that the SDL declaration refers to an inherited property that is being overloaded (by adding more constraints, for example), the *overloaded* keyword must be used. This is to prevent unintentional overloading due to name clashes:

```
abstract type Named {
    property name -> str;
}

type User extending Named {
    # define concrete properties
    overloaded required property name -> str;
    # ... other links and properties
}
```

```
abstract type Named {
    name: str;
```

(continues on next page)

(continued from previous page)

```

}

type User extending Named {
    # define concrete properties
    overloaded required name: str;
    # ... other links and properties
}

```

### 8.2.5.2 Syntax

Define a new property corresponding to the *more explicit DDL commands*.

```

# Concrete property form used inside type declaration:
[ overloaded ] [{required | optional}] [{single | multi}]
property <name>
[ extending <base> [, ...] ] -> <type>
[ "{"
    [ default := <expression> ; ]
    [ readonly := {true | false} ; ]
    [ <annotation-declarations> ]
    [ <constraint-declarations> ]
    ...
}
]

# Computed property form used inside type declaration:
[{required | optional}] [{single | multi}]
property <name> := <expression>;

# Computed property form used inside type declaration (extended):
[ overloaded ] [{required | optional}] [{single | multi}]
property <name>
[ extending <base> [, ...] ] [-> <type>]
[ "{"
    [ using (<expression>) ;
    [ <annotation-declarations> ]
    [ <constraint-declarations> ]
    ...
}
]

# Abstract property form:
abstract property [<module>::]<name> [extending <base> [, ...]]
[ "{"
    [ readonly := {true | false} ; ]
    [ <annotation-declarations> ]
    ...
}
]

```

```

# Concrete property form used inside type declaration:
[ overloaded ] [{required | optional}] [{single | multi}]
[ property ] <name> : <type>
[ "{"

```

(continues on next page)

(continued from previous page)

```

[ extending <base> [, ...] ; ]
[ default := <expression> ; ]
[ readonly := {true | false} ; ]
[ <annotation-declarations> ]
[ <constraint-declarations> ]

...
"}" ]

# Computed property form used inside type declaration:
[required | optional] [{single | multi}]
property <name> := <expression>;

# Computed property form used inside type declaration (extended):
[ overloaded ] [required | optional] [{single | multi}]
property <name> [: <type>]
[ "{"
  using (<expression>) ;
  [ extending <base> [, ...] ; ]
  [ <annotation-declarations> ]
  [ <constraint-declarations> ]
  ...
"}" ]

# Abstract property form:
abstract property [<module>::]<name>
[ "{"
  [ extending <base> [, ...] ; ]
  [ readonly := {true | false} ; ]
  [ <annotation-declarations> ]
  ...
"}" ]

```

### 8.2.5.3 Description

There are several forms of property declaration, as shown in the syntax synopsis above. The first form is the canonical definition form, the second and third forms are used for defining a *computed property*, and the last one is a form to define an *abstract property*. The abstract form allows declaring the property directly inside a *module*. Concrete property forms are always used as sub-declarations for an *object type* or a *link*.

The following options are available:

**overloaded** If specified, indicates that the property is inherited and that some feature of it may be altered in the current object type. It is an error to declare a property as *overloaded* if it is not inherited.

**required** If specified, the property is considered *required* for the parent object type. It is an error for an object to have a required property resolve to an empty value. Child properties **always** inherit the *required* attribute, i.e it is not possible to make a required property non-required by extending it.

**optional** This is the default qualifier assumed when no qualifier is specified, but it can also be specified explicitly. The property is considered *optional* for the parent object type, i.e. it is possible for the property to resolve to an empty value.

**multi** Specifies that there may be more than one instance of this property in an object, in other words, `Object.property` may resolve to a set of a size greater than one.

**single** Specifies that there may be at most *one* instance of this property in an object, in other words, `Object.property` may resolve to a set of a size not greater than one. `single` is assumed if neither `multi` nor `single` qualifier is specified.

**extending <base> [, ...]** Optional clause specifying the *parents* of the new property item.

Use of `extending` creates a persistent schema relationship between the new property and its parents. Schema modifications to the parent(s) propagate to the child.

As of EdgeDB 3.0, the `extended` clause is now a sub-declaration of the property and included inside the curly braces rather than an option as in earlier versions.

**<type>** The type must be a valid *type expression* denoting a non-abstract scalar or a container type.

The valid SDL sub-declarations are listed below:

**default := <expression>** Specifies the default value for the property as an EdgeQL expression. The default value is used in an `insert` statement if an explicit value for this property is not specified.

**readonly := {true | false}** If true, the property is considered *read-only*. Modifications of this property are prohibited once an object is created. All of the derived properties **must** preserve the original *read-only* value.

**<annotation-declarations>** Set property *annotation* to a given *value*.

**<constraint-declarations>** Define a concrete *constraint* on the property.

See also
<i>Schema &gt; Properties</i>
<i>DDL &gt; Properties</i>
<i>Introspection &gt; Object types</i>

## 8.2.6 Expression Aliases

This section describes the SDL declarations pertaining to *expression aliases*.

### 8.2.6.1 Example

Declare a “UserAlias” that provides additional information for a “User” via a *computed link* “friend\_of”:

```
alias UserAlias := User {
    # declare a computed link
    friend_of := User.<friends[is User]
};
```

### 8.2.6.2 Syntax

Define a new alias corresponding to the *more explicit DDL commands*.

```
alias <alias-name> := <alias-expr> ;

alias <alias-name> "{"
    using <alias-expr>;
    [ <annotation-declarations> ]
}" ;
```

### 8.2.6.3 Description

This declaration defines a new alias with the following options:

**<alias-name>** The name (optionally module-qualified) of an alias to be created.

**<alias-expr>** The aliased expression. Can be any valid EdgeQL expression.

The valid SDL sub-declarations are listed below:

**<annotation-declarations>** Set alias *annotation* to a given *value*.

See also
<i>Schema &gt; Aliases</i>
<i>DDL &gt; Aliases</i>
<i>Cheatsheets &gt; Aliases</i>

## 8.2.7 Indexes

This section describes the SDL declarations pertaining to *indexes*.

### 8.2.7.1 Example

Declare an index for a “User” based on the “name” property:

```
type User {
    required property name -> str;
    property address -> str;

    multi link friends -> User;

    # define an index for User based on name
    index on (.name) {
        annotation title := 'User name index';
    }
}
```

```
type User {
    required name: str;
    address: str;

    multi friends: User;

    # define an index for User based on name
    index on (.name) {
        annotation title := 'User name index';
    }
}
```

### 8.2.7.2 Syntax

Define a new index corresponding to the [more explicit DDL commands](#).

```
index on ( <index-expr> )
[ except ( <exception-expr> ) ]
[ "{" <annotation-declarations> "}" ] ;
```

### 8.2.7.3 Description

This declaration defines a new index with the following options:

**on** ( <index-expr> ) The specific expression for which the index is made. Note also that <index-expr> itself has to be parenthesized.

**except** ( <exception-expr> )

An optional expression defining a condition to create exceptions to the index. If <exception-expr> evaluates to `true`, the object is omitted from the index. If it evaluates to `false` or `{}`, it appears in the index.

The valid SDL sub-declarations are listed below:

**<annotation-declarations>** Set index [annotation](#) to a given *value*.

See also
<a href="#">Schema &gt; Indexes</a>
<a href="#">DDL &gt; Indexes</a>
<a href="#">Introspection &gt; Indexes</a>

## 8.2.8 Constraints

This section describes the SDL declarations pertaining to [constraints](#).

### 8.2.8.1 Examples

Declare an *abstract* constraint:

```
abstract constraint min_value(min: anytype) {
    errmessage := 
        'Minimum allowed value for {__subject__} is {min}.';
    using (__subject__ >= min);
}
```

Declare a *concrete* constraint on an integer type:

```
scalar type posint64 extending int64 {
    constraint min_value(0);
}
```

Declare a *concrete* constraint on an object type:

```
type Vector {
    required property x -> float64;
    required property y -> float64;
    constraint expression on (
        __subject__.x^2 + __subject__.y^2 < 25
    );
}
```

```
type Vector {
    required x: float64;
    required y: float64;
    constraint expression on (
        __subject__.x^2 + __subject__.y^2 < 25
    );
}
```

### 8.2.8.2 Syntax

Define a constraint corresponding to the *more explicit DDL commands*.

```
[{abstract | delegated}] constraint <name> [ ( [<argspec>] [, ...] ) ]
    [ on ( <subject-expr> ) ]
    [ except ( <except-expr> ) ]
    [ extending <base> [, ...] ]
"{
    [ using <constr-expression> ; ]
    [ errmessage := <error-message> ; ]
    [ <annotation-declarations> ]
    [ ... ]
}"
# where <argspec> is:

[ <argname>: ] {<argtype> | <argvalue>}
```

### 8.2.8.3 Description

This declaration defines a new constraint with the following options:

**abstract** If specified, the constraint will be *abstract*.

**delegated** If specified, the constraint is defined as *delegated*, which means that it will not be enforced on the type it's declared on, and the enforcement will be delegated to the subtypes of this type. This is particularly useful for *exclusive* constraints in abstract types. This is only valid for *concrete constraints*.

**<name>** The name (optionally module-qualified) of the new constraint.

**<argspec>** An optional list of constraint arguments.

For an *abstract constraint* **<argname>** optionally specifies the argument name and **<argtype>** specifies the argument type.

For a *concrete constraint* **<argname>** optionally specifies the argument name and **<argvalue>** specifies the argument value. The argument value specification must match the parameter declaration of the abstract constraint.

**on ( <subject-expr> )** An optional expression defining the *subject* of the constraint. If not specified, the subject is the value of the schema item on which the concrete constraint is defined. The expression must refer to the original subject of the constraint as `__subject__`. Note also that `<subject-expr>` itself has to be parenthesized.

**except ( <exception-expr> )**

An optional expression defining a condition to create exceptions to the constraint. If `<exception-expr>` evaluates to `true`, the constraint is ignored for the current subject. If it evaluates to `false` or `{}`, the constraint applies normally.

`except` may only be declared on object constraints, and is otherwise follows the same rules as `on`, above.

**extending <base> [, ...]** If specified, declares the *parent* constraints for this abstract constraint.

The valid SDL sub-declarations are listed below:

**using <constr\_expression>** A boolean expression that returns `true` for valid data and `false` for invalid data.

The expression may refer to the subject of the constraint as `__subject__`. This declaration is only valid for *abstract constraints*.

**errmessage := <error\_message>** An optional string literal defining the error message template that is raised when the constraint is violated. The template is a formatted string that may refer to constraint context variables in curly braces. The template may refer to the following:

- `$argname` – the value of the specified constraint argument
- `__subject__` – the value of the `title` annotation of the scalar type, property or link on which the constraint is defined.

If the content of curly braces does not match any variables, the curly braces are emitted as-is. They can also be escaped by using double curly braces.

**<annotation-declarations>** Set constraint `annotation` to a given *value*.

See also
<a href="#">Schema &gt; Constraints</a>
<a href="#">DDL &gt; Constraints</a>
<a href="#">Introspection &gt; Constraints</a>
<a href="#">Standard Library &gt; Constraints</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; Constraints</a>

## 8.2.9 Annotations

This section describes the SDL declarations pertaining to *annotations*.

### 8.2.9.1 Examples

Declare a new annotation:

```
abstract annotation admin_note;
```

Specify the value of an annotation for a type:

```
type Status {
    annotation admin_note := 'system-critical';
    required property name -> str {
        constraint exclusive
```

(continues on next page)

(continued from previous page)

```

    }
}
```

```

type Status {
    annotation admin_note := 'system-critical';
    required name: str {
        constraint exclusive
    }
}
```

### 8.2.9.2 Syntax

Define a new annotation corresponding to the *more explicit DDL commands*.

```

# Abstract annotation form:
abstract [ inheritable ] annotation <name>
[ "{" <annotation-declarations>; [...] "}" ] ;

# Concrete annotation (same as <annotation-declarations>) form:
annotation <name> := <value> ;
```

### 8.2.9.3 Description

There are two forms of annotation declarations: abstract and concrete. The *abstract annotation* form is used for declaring new kinds of annotation in a module. The *concrete annotation* declarations are used as sub-declarations for all other declarations in order to actually annotate them.

The annotation declaration options are as follows:

**abstract** If specified, the annotation will be *abstract*.

**inheritable** If specified, the annotation will be *inheritable*. The annotations are non-inheritable by default. That is, if a schema item has an annotation defined on it, the descendants of that schema item will not automatically inherit the annotation. Normal inheritance behavior can be turned on by declaring the annotation with the **inheritable** qualifier. This is only valid for *abstract annotation*.

**<name>** The name (optionally module-qualified) of the annotation.

**<value>** Any string value that the specified annotation is intended to have for the given context.

The only valid SDL sub-declarations are *concrete annotations*:

**<annotation-declarations>** Annotations can also have annotations. Set the *annotation* of the enclosing annotation to a specific value.

See also
<i>Schema &gt; Annotations</i>
<i>DDL &gt; Annotations</i>
<i>Cheatsheets &gt; Annotations</i>
<i>Introspection &gt; Object types</i>

## 8.2.10 Globals

This section describes the SDL commands pertaining to global variables.

### 8.2.10.1 Examples

Declare a new global variable:

```
global current_user_id -> uuid;
global current_user := (
    select User filter .id = global current_user_id
);
```

Set the global variable to a specific value using *session-level commands*:

```
set global current_user_id :=
<uuid>'00ea8eaa-02f9-11ed-a676-6bd11cc6c557';
```

Use the computed global variable that is based on the value that was just set:

```
select global current_user { name };
```

*Reset* the global variable to its default value:

```
reset global user_id;
```

### 8.2.10.2 Syntax

Define a new global variable corresponding to the *more explicit DDL commands*.

```
# Global variable declaration:
[ { required | optional } ] [ single ]
    global <name> -> <type>
    [
        "{"
            [ default := <expression> ; ]
            [ <annotation-declarations> ]
            ...
        "}"
    ]

# Computed global variable declaration:
[ { required | optional } ] [ { single | multi } ]
    global <name> := <expression>;
```

### 8.2.10.3 Description

There are two different forms of `global` declaration, as shown in the syntax synopsis above. The first form is for defining a global variable that can be `set` in a session. The second form is not directly set, but instead it is *computed* based on an expression, potentially deriving its value from other global variables.

The following options are available:

**required** If specified, the global variable is considered *required*. It is an error for this variable to have an empty value. If a global variable is declared *required*, it must also declare a `default` value.

**optional** This is the default qualifier assumed when no qualifier is specified, but it can also be specified explicitly. The global variable is considered *optional*, i.e. it is possible for the variable to have an empty value.

**multi** Specifies that the global variable may have a set of values. Only *computed* global variables can have this qualifier.

**single** Specifies that the global variable must have at most a *single* value. It is assumed that a global variable is *single* if neither `multi` nor `single` qualifier is specified. All non-computed global variables must be *single*.

**<name>** Specifies the name of the global variable. The name has to be either fully-qualified with the module name it belongs to or it will be assumed to belong to the module in which it appears.

**<type>** The type must be a valid `type expression` denoting a non-abstract scalar or a container type.

**<name> := <expression>** Defines a *computed* global variable. The provided expression can be any valid EdgeQL expression, including one referring to other global variables. The type of a *computed* global variable is not limited to scalar and container types, but also includes object types. So it is possible to use that to define a global object variable based on another global scalar variable.

For example:

```
# Global scalar variable that can be set in a session:  
global current_user_id -> uuid;  
# Global computed object based on that:  
global current_user := (  
    select User filter .id = global current_user_id  
)
```

The valid SDL sub-declarations are listed below:

**default := <expression>** Specifies the default value for the global variable as an EdgeQL expression. The default value is used by the session if the value was not explicitly specified or by the client or was reset with the `reset` command.

**<annotation-declarations>** Set global variable `annotation` to a given `value`.

See also
<a href="#">Schema &gt; Globals</a>
<a href="#">DDL &gt; Globals</a>

## 8.2.11 Access Policies

This section describes the SDL declarations pertaining to access policies.

### 8.2.11.1 Examples

Declare a schema where users can only see their own profiles:

```
# Declare some global variables to store "current user"
# information.
global current_user_id -> uuid;
global current_user := (
    select User filter .id = global current_user_id
);

type User {
    required property name -> str;
}

type Profile {
    link owner -> User;

    # Only allow reading to the owner, but also
    # ensure that a user cannot set the "owner" link
    # to anything but themselves.
    access policy owner_only
        allow all using (.owner = global current_user)
        { errmessage := 'Profile may only be accessed by the owner'; }
}
```

```
# Declare some global variables to store "current user"
# information.
global current_user_id: uuid;
global current_user := (
    select User filter .id = global current_user_id
);

type User {
    required name: str;
}

type Profile {
    owner: User;

    # Only allow reading to the owner, but also
    # ensure that a user cannot set the "owner" link
    # to anything but themselves.
    access policy owner_only
        allow all using (.owner = global current_user);
}
```

### 8.2.11.2 Syntax

Define a new access policy corresponding to the *more explicit DDL commands*.

```
# Access policy used inside a type declaration:  
access policy <name>  
  [ when (<condition>) ]  
  { allow | deny } <action> [, <action> ... ]  
  [ using (<expr>) ]  
  [ <annotation-declarations> ] ;  
  
# where <action> is one of  
all  
select  
insert  
delete  
update [{ read | write }]
```

```
# Access policy used inside a type declaration:  
access policy <name>  
  [ when (<condition>) ]  
  { allow | deny } <action> [, <action> ... ]  
  [ using (<expr>) ]  
  [ <annotation-declarations> ]  
  [ "{"  
    [ errmessage := value ; ]  
  "}" ] ;  
  
# where <action> is one of  
all  
select  
insert  
delete  
update [{ read | write }]
```

### 8.2.11.3 Description

Access policies are used to implement object-level security and as such they are defined on object types. In practice the access policies often work together with *global variables*.

Access policies are an opt-in feature, so once at least one access policy is defined for a given type, all access not explicitly allowed by that policy becomes forbidden.

Any sub-type *extending* a base type also inherits all the access policies of the base type.

The access policy declaration options are as follows:

**<name>** The name of the access policy.

**when (<condition>)** Specifies which objects this policy applies to. The <condition> has to be a *bool* expression.

When omitted, it is assumed that this policy applies to all objects of a given type.

**allow** Indicates that qualifying objects should allow access under this policy.

**deny** Indicates that qualifying objects should *not* allow access under this policy. This flavor supersedes any **allow** policy and can be used to selectively deny access to a subset of objects that otherwise explicitly allows accessing them.

**all** Apply the policy to all actions. It is exactly equivalent to listing **select**, **insert**, **delete**, **update** actions explicitly.

**select** Apply the policy to all selection queries. Note that any object that cannot be selected, cannot be modified either. This makes **select** the most basic “visibility” policy.

**insert** Apply the policy to all inserted objects. If a newly inserted object would violate this policy, an error is produced instead.

**delete** Apply the policy to all objects about to be deleted. If an object does not allow access under this kind of policy, it is not going to be considered by any **delete** command.

Note that any object that cannot be selected, cannot be modified either.

**update read** Apply the policy to all objects selected for an update. If an object does not allow access under this kind of policy, it is not visible cannot be updated.

Note that any object that cannot be selected, cannot be modified either.

**update write** Apply the policy to all objects at the end of an update. If an updated object violates this policy, an error is produced instead.

Note that any object that cannot be selected, cannot be modified either.

**update** This is just a shorthand for **update read** and **update write**.

Note that any object that cannot be selected, cannot be modified either.

**using <expr>** Specifies what the policy is with respect to a given eligible (based on **when** clause) object. The **<expr>** has to be a **bool** expression. The specific meaning of this value also depends on whether this policy flavor is **allow** or **deny**.

When omitted, it is assumed that this policy applies to all eligible objects of a given type.

**set errmessage := <value>** Set a custom error message of **<value>** that is displayed when this access policy prevents a write action.

**<annotation-declarations>** Set access policy **annotation** to a given **value**.

See also
<a href="#">Schema &gt; Access policies</a>
<a href="#">DDL &gt; Access policies</a>

## 8.2.12 Functions

This section describes the SDL declarations pertaining to *functions*.

### 8.2.12.1 Example

Declare a custom function that concatenates the length of a string to the end of the that string:

```
function foo(s: str) -> str
    using (
        select s ++ <str>len(a)
    );
```

### 8.2.12.2 Syntax

Define a new function corresponding to the *more explicit DDL commands*.

```
function <name> ([ <argspec> ] [ , ... ]) -> <returnspec>
using ( <edgeql> );

function <name> ([ <argspec> ] [ , ... ]) -> <returnspec>
using <language> <functionbody> ;

function <name> ([ <argspec> ] [ , ... ]) -> <returnspec>
"""
[ <annotation-declarations> ]
[ volatility := {'Immutable' | 'Stable' | 'Volatile'} ]
[ using ( <expr> ) ; ]
[ using <language> <functionbody> ; ]
[ ... ]
""" ;

# where <argspec> is:

[ <argkind> ] <argname>: [ <typequal> ] <argtype> [ = <default> ]

# <argkind> is:

[ { variadic | named only } ]

# <typequal> is:

[ { set of | optional } ]

# and <returnspec> is:

[ <typequal> ] <rettype>
```

### 8.2.12.3 Description

This declaration defines a new constraint with the following options:

**<name>** The name (optionally module-qualified) of the function to create.

**<argkind>** The kind of an argument: `variadic` or `named only`.

If not specified, the argument is called *positional*.

The `variadic` modifier indicates that the function takes an arbitrary number of arguments of the specified type. The passed arguments will be passed as an array of the argument type. Positional arguments cannot follow a `variadic` argument. `variadic` parameters cannot have a default value.

The `named only` modifier indicates that the argument can only be passed using that specific name. Positional arguments cannot follow a `named only` argument.

**<argname>** The name of an argument. If `named only` modifier is used this argument *must* be passed using this name only.

**<typequal>** The type qualifier: `set of` or `optional`.

The `set of` qualifier indicates that the function is taking the argument as a *whole set*, as opposed to being called on the input product element-by-element.

The `optional` qualifier indicates that the function will be called if the argument is an empty set. The default behavior is to return an empty set if the argument is not marked as `optional`.

**<argtype>** The data type of the function's arguments (optionally module-qualified).

**<default>** An expression to be used as default value if the parameter is not specified. The expression has to be of a type compatible with the type of the argument.

**<rettype>** The return data type (optionally module-qualified).

The `set of` modifier indicates that the function will return a non-singleton set.

The `optional` qualifier indicates that the function may return an empty set.

The valid SDL sub-declarations are listed below:

**volatility := {'Immutable' | 'Stable' | 'Volatile'}** Function volatility determines how aggressively the compiler can optimize its invocations.

If not explicitly specified the function volatility is set to `Volatile` by default.

- A `Volatile` function can modify the database and can return different results on successive calls with the same arguments.
- A `Stable` function cannot modify the database and is guaranteed to return the same results given the same arguments *within a single statement*.
- An `Immutable` function cannot modify the database and is guaranteed to return the same results given the same arguments *forever*.

**using ( <expr> )** Specified the body of the function. `<expr>` is an arbitrary EdgeQL expression.

**using <language> <functionbody>** A verbose version of the `using` clause that allows to specify the language of the function body.

- `<language>` is the name of the language that the function is implemented in. Currently can only be `edgeql`.
- `<functionbody>` is a string constant defining the function. It is often helpful to use *dollar quoting* to write the function definition string.

**<annotation-declarations>** Set function `annotation` to a given `value`.

The function name must be distinct from that of any existing function with the same argument types in the same module. Functions of different argument types can share a name, in which case the functions are called *overloaded functions*.

See also
<a href="#">Schema &gt; Functions</a>
<a href="#">DDL &gt; Functions</a>
<a href="#">Reference &gt; Function calls</a>
<a href="#">Introspection &gt; Functions</a>
<a href="#">Cheatsheets &gt; Functions</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; User-Defined Functions</a>

## 8.2.13 Triggers

This section describes the SDL declarations pertaining to *triggers*.

### 8.2.13.1 Example

Declare a trigger that inserts a Log object for each new User object:

```
type User {
    required name: str;

    trigger log_insert after insert for each do (
        insert Log {
            action := 'insert',
            target_name := __new__.name
        }
    );
}
```

### 8.2.13.2 Syntax

Define a new trigger corresponding to the *more explicit DDL commands*.

```
type <type-name> "{"
    trigger <name>
        after
            {insert | update | delete} [, ...]
            for {each | all}
            do <expr>
    "}"
```

### 8.2.13.3 Description

This declaration defines a new trigger with the following options:

**<type-name>** The name (optionally module-qualified) of the type to be triggered on.

**<name>** The name of the trigger.

**insert | update | delete [, ...]** The query type (or types) to trigger on. Separate multiple values with commas to invoke the same trigger for multiple types of queries.

**each** The expression will be evaluated once per modified object. `__new__` and `__old__` in this context within the expression will refer to a single object.

**all** The expression will be evaluated once for the entire query, even if multiple objects were modified. `__new__` and `__old__` in this context within the expression refer to sets of the modified objects.

**<expr>** The expression to be evaluated when the trigger is invoked.

The trigger name must be distinct from that of any existing trigger on the same type.

See also
<a href="#">Schema &gt; Triggers</a>
<a href="#">DDL &gt; Triggers</a>
<a href="#">Introspection &gt; Triggers</a>

### 8.2.14 Mutation rewrites

This section describes the SDL declarations pertaining to *mutation rewrites*.

#### 8.2.14.1 Example

Declare two mutation rewrites: one that sets a `created` property when a new object is inserted and one that sets a `modified` property on each update:

```
type User {
    created: datetime {
        rewrite insert using (datetime_of_statement());
    }
    modified: datetime {
        rewrite update using (datetime_of_statement());
    }
};
```

#### 8.2.14.2 Syntax

Define a new mutation rewrite corresponding to the *more explicit DDL commands*.

```
rewrite {insert | update} [, ...]
    using <expr>
```

Mutation rewrites must be defined inside a property or link block.

### 8.2.14.3 Description

This declaration defines a new trigger with the following options:

**insert | update [, ...]** The query type (or types) the rewrite runs on. Separate multiple values with commas to invoke the same rewrite for multiple types of queries.

**<expr>** The expression to be evaluated to produce the new value of the property.

See also
<a href="#">Schema &gt; Mutation rewrites</a>
<a href="#">DDL &gt; Mutation rewrites</a>
<a href="#">Introspection &gt; Mutation rewrites</a>

## 8.2.15 Extensions

This section describes the SDL commands pertaining to *extensions*.

### 8.2.15.1 Syntax

Declare that the current schema enables a particular extension.

```
using extension <ExtensionName> ";"
```

### 8.2.15.2 Description

Extension declaration must be outside any *module block* since extensions affect the entire database and not a specific module.

### 8.2.15.3 Examples

Enable *GraphQL* extension for the current schema:

```
using extension graphql;
```

Enable *EdgeQL over HTTP* extension for the current database:

```
using extension edgeql_http;
```

## 8.2.16 Future Behavior

This section describes the SDL commands pertaining to *future*.

### 8.2.16.1 Syntax

Declare that the current schema enables a particular future behavior.

```
using future <FutureBehavior> ";"
```

### 8.2.16.2 Description

Future behavior declaration must be outside any *module block* since this behavior affects the entire database and not a specific module.

### 8.2.16.3 Examples

Enable simpler non-recursive access policy behavior *non-recursive access policy* for the current schema:

```
using extension nonrecursive_access_policies;
```

## 8.3 DDL

### 8.3.1 Modules

This section describes the DDL commands pertaining to *modules*.

#### 8.3.1.1 Create module

##### eql-statement

Create a new module.

```
create module <name> [ if not exists ];
```

There's a *corresponding SDL declaration* for a module, although in SDL a module declaration is likely to also include that module's content.

You may also create a nested module.

```
create module <parent-name>::<name> [ if not exists ];
```

#### 8.3.1.1.1 Description

The command `create module` defines a new module for the current database. The name of the new module must be distinct from any existing module in the current database. Unlike *SDL module declaration* the `create module` command does not have sub-commands, as module contents are created separately.

### 8.3.1.1.2 Parameters

**if not exists** Normally creating a module that already exists is an error, but with this flag the command will succeed. It is useful for scripts that add something to a module or if the module is missing the module is created as well.

### 8.3.1.1.3 Examples

Create a new module:

```
create module payments;
```

Create a new nested module:

```
create module payments::currencies;
```

## 8.3.1.2 Drop module

### eql-statement

Remove a module.

```
drop module <name> ;
```

### 8.3.1.2.1 Description

The command `drop module` removes an existing module from the current database. All schema items and data contained in the module are removed as well.

### 8.3.1.2.2 Examples

Remove a module:

```
drop module payments;
```

## 8.3.2 Object Types

This section describes the DDL commands pertaining to *object types*.

### 8.3.2.1 Create type

**eql-statement**

**eql-haswith**

*Define* a new object type.

```
[ with <with-item> [, ...] ]
create [abstract] type <name> [ extending <supertype> [, ...] ]
[ "{" <subcommand>; [...] "}" ] ;

# where <subcommand> is one of

create annotation <annotation-name> := <value>
create link <link-name> ...
create property <property-name> ...
create constraint <constraint-name> ...
create index on <index-expr>
```

#### 8.3.2.1.1 Description

The command `create type` defines a new object type for use in the current database.

If *name* is qualified with a module name, then the type is created in that module, otherwise it is created in the current module. The type name must be distinct from that of any existing schema item in the module.

#### 8.3.2.1.2 Parameters

Most sub-commands and options of this command are identical to the *SDL object type declaration*, with some additional features listed below:

**with <with-item> [, ...]** Alias declarations.

The `with` clause allows specifying module aliases that can be referenced by the command. See [With block](#) for more information.

The following subcommands are allowed in the `create type` block:

**create annotation <annotation-name> := <value>** Set object type *<annotation-name>* to *<value>*.

See [create annotation](#) for details.

**create link <link-name> ...** Define a new link for this object type. See [create link](#) for details.

**create property <property-name> ...** Define a new property for this object type. See [create property](#) for details.

**create constraint <constraint-name> ...** Define a concrete constraint for this object type. See [create constraint](#) for details.

**create index on <index-expr>** Define a new *index* using *index-expr* for this object type. See [create index](#) for details.

### 8.3.2.1.3 Examples

Create an object type User:

```
create type User {  
    create property name -> str;  
};
```

### 8.3.2.2 Alter type

eql-statement

eql-haswith

Change the definition of an *object type*.

```
[ with <with-item> [, ...] ]  
alter type <name>  
[ "{" <subcommand>; [...] "}" ] ;  
  
[ with <with-item> [, ...] ]  
alter type <name> <subcommand> ;  
  
# where <subcommand> is one of  
  
    rename to <newname>  
    extending <parent> [, ...]  
    create annotation <annotation-name> := <value>  
    alter annotation <annotation-name> := <value>  
    drop annotation <annotation-name>  
    create link <link-name> ...  
    alter link <link-name> ...  
    drop link <link-name> ...  
    create property <property-name> ...  
    alter property <property-name> ...  
    drop property <property-name> ...  
    create constraint <constraint-name> ...  
    alter constraint <constraint-name> ...  
    drop constraint <constraint-name> ...  
    create index on <index-expr>  
    drop index on <index-expr>
```

#### 8.3.2.2.1 Description

The command `alter type` changes the definition of an object type. `name` must be a name of an existing object type, optionally qualified with a module name.

### 8.3.2.2.2 Parameters

The following subcommands are allowed in the `alter type` block:

**with <with-item> [, ...]** Alias declarations.

The `with` clause allows specifying module aliases that can be referenced by the command. See [With block](#) for more information.

**<name>** The name (optionally module-qualified) of the type being altered.

**extending <parent> [, ...]** Alter the supertype list. The full syntax of this subcommand is:

```
extending <parent> [, ...]
  [first | last | before <exparent> | after <exparent> ]
```

This subcommand makes the type a subtype of the specified list of supertypes. The requirements for the parent-child relationship are the same as when creating an object type.

It is possible to specify the position in the parent list using the following optional keywords:

- `first` – insert parent(s) at the beginning of the parent list,
- `last` – insert parent(s) at the end of the parent list,
- `before <parent>` – insert parent(s) before an existing `parent`,
- `after <parent>` – insert parent(s) after an existing `parent`.

**alter annotation <annotation-name>;** Alter object type annotation `<annotation-name>`. See [alter annotation](#) for details.

**drop annotation <annotation-name>** Remove object type `<annotation-name>`. See [drop annotation](#) for details.

**alter link <link-name> ...** Alter the definition of a link for this object type. See [alter link](#) for details.

**drop link <link-name>** Remove a link item from this object type. See [drop link](#) for details.

**alter property <property-name> ...** Alter the definition of a property item for this object type. See [alter property](#) for details.

**drop property <property-name>** Remove a property item from this object type. See [drop property](#) for details.

**alter constraint <constraint-name> ...** Alter the definition of a constraint for this object type. See [alter constraint](#) for details.

**drop constraint <constraint-name>;** Remove a constraint from this object type. See [drop constraint](#) for details.

**drop index on <index-expr>** Remove an `index` defined as `index-expr` from this object type. See [drop index](#) for details.

All the subcommands allowed in the `create type` block are also valid subcommands for `alter type` block.

### 8.3.2.2.3 Examples

Alter the User object type to make name required:

```
alter type User {  
    alter property name {  
        set required;  
    }  
};
```

### 8.3.2.3 Drop type

**eql-statement**

**eql-haswith**

Remove the specified object type from the schema.

```
drop type <name> ;
```

#### 8.3.2.3.1 Description

The command `drop type` removes the specified object type from the schema. All subordinate schema items defined on this type, such as links and indexes, are removed as well.

#### 8.3.2.3.2 Examples

Remove the User object type:

```
drop type User;
```

See also
<a href="#">Schema &gt; Object types</a>
<a href="#">SDL &gt; Object types</a>
<a href="#">Introspection &gt; Object types</a>
<a href="#">Cheatsheets &gt; Object types</a>

## 8.3.3 Scalar Types

This section describes the DDL commands pertaining to *scalar types*.

### 8.3.3.1 Create scalar type

**eql-statement**

**eql-haswith**

*Define* a new scalar type.

```
[ with <with-item> [, ...] ]
create [ abstract ] scalar type <name> [ extending <supertype> ]
[ "{" <subcommand>; [...] "}" ] ;

# where <subcommand> is one of

create annotation <annotation-name> := <value>
create constraint <constraint-name> ...
```

#### 8.3.3.1.1 Description

The command `create scalar type` defines a new scalar type for use in the current database.

If *name* is qualified with a module name, then the type is created in that module, otherwise it is created in the current module. The type name must be distinct from that of any existing schema item in the module.

If the `abstract` keyword is specified, the created type will be *abstract*.

All non-abstract scalar types must have an underlying core implementation. For user-defined scalar types this means that `create scalar type` must have another non-abstract scalar type as its *supertype*.

The most common use of `create scalar type` is to define a scalar subtype with constraints.

Most sub-commands and options of this command are identical to the [SDL scalar type declaration](#). The following subcommands are allowed in the `create scalar type` block:

**create annotation** <annotation-name> := <value>; Set scalar type's <annotation-name> to <value>.

See `create annotation` for details.

**create constraint** <constraint-name> ... Define a new constraint for this scalar type. See `create constraint` for details.

#### 8.3.3.1.2 Examples

Create a new non-negative integer type:

```
create scalar type posint64 extending int64 {
    create constraint min_value(0);
};
```

Create a new enumerated type:

```
create scalar type Color
extending enum<Black, White, Red>;
```

### 8.3.3.2 Alter scalar type

**eql-statement**

**eql-haswith**

Alter the definition of a *scalar type*.

```
[ with <with-item> [, ...] ]
alter scalar type <name>
"{" <subcommand>; [...] "}" ;

# where <subcommand> is one of

rename to <newname>
extending ...
create annotation <annotation-name> := <value>
alter annotation <annotation-name> := <value>
drop annotation <annotation-name>
create constraint <constraint-name> ...
alter constraint <constraint-name> ...
drop constraint <constraint-name> ...
```

#### 8.3.3.2.1 Description

The command `alter scalar type` changes the definition of a scalar type. *name* must be a name of an existing scalar type, optionally qualified with a module name.

The following subcommands are allowed in the `alter scalar type` block:

**rename to** <*newname*>; Change the name of the scalar type to *newname*.

**extending** ... Alter the supertype list. It works the same way as in [alter type](#).

**alter annotation** <*annotation-name*>; Alter scalar type <*annotation-name*>. See [alter annotation](#) for details.

**drop annotation** <*annotation-name*> Remove scalar type's <*annotation-name*> from <*value*>. See [drop annotation](#) for details.

**alter constraint** <*constraint-name*> ... Alter the definition of a constraint for this scalar type. See [alter constraint](#) for details.

**drop constraint** <*constraint-name*> Remove a constraint from this scalar type. See [drop constraint](#) for details.

All the subcommands allowed in the `create scalar type` block are also valid subcommands for `alter scalar type` block.

### 8.3.3.2.2 Examples

Define a new constraint on a scalar type:

```
alter scalar type posint64 {
    create constraint max_value(100);
};
```

Add one more label to an enumerated type:

```
alter scalar type Color
    extending enum<Black, White, Red, Green>;
```

### 8.3.3.3 Drop scalar type

**eql-statement**

**eql-haswith**

Remove a scalar type.

```
[ with <with-item> [, ...] ]
drop scalar type <name> ;
```

#### 8.3.3.3.1 Description

The command `drop scalar type` removes a scalar type.

#### 8.3.3.3.2 Parameters

***name*** The name (optionally qualified with a module name) of an existing scalar type.

#### 8.3.3.3.3 Example

Remove a scalar type:

```
drop scalar type posint64;
```

## 8.3.4 Links

This section describes the DDL commands pertaining to [links](#).

### 8.3.4.1 Create link

eql-statement

eql-haswith

*Define* a new link.

```
[ with <with-item> [, ...] ]
{create|alter} type <TypeName> "{"
    [ ... ]
    create [{required | optional}]
        [{single | multi}]
            link <name>
                [ extending <base> [, ...] ] -> <type>
                [ "{" <subcommand>; [...] "}" ] ;
    [ ... ]
}"
# Computed link form:

[ with <with-item> [, ...] ]
{create|alter} type <TypeName> "{"
    [ ... ]
    create [{required | optional}]
        [{single | multi}]
            link <name> := <expression>;
    [ ... ]
}"
# Abstract link form:

[ with <with-item> [, ...] ]
create abstract link [<module>::]<name> [extending <base> [, ...]]
[ "{" <subcommand>; [...] "}" ]

# where <subcommand> is one of

set default := <expression>
set readonly := {true | false}
create annotation <annotation-name> := <value>
create property <property-name> ...
create constraint <constraint-name> ...
on target delete <action>
reset on target delete
create index on <index-expr>
```

### 8.3.4.1.1 Description

The combinations of `create type ... create link` and `alter type ... create link` define a new concrete link for a given object type.

There are three forms of `create link`, as shown in the syntax synopsis above. The first form is the canonical definition form, the second form is a syntax shorthand for defining a *computed link*, and the third is a form to define an abstract link item. The abstract form allows creating the link in the specified `<module>`. Concrete link forms are always created in the same module as the containing object type.

### 8.3.4.1.2 Parameters

Most sub-commands and options of this command are identical to the *SDL link declaration*. The following subcommands are allowed in the `create link` block:

`set default := <expression>` Specifies the default value for the link as an EdgeQL expression. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

`set readonly := {true | false}` Specifies whether the link is considered *read-only*. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

`create annotation <annotation-name> := <value>;` Add an annotation `<annotation-name>` set to `<value>` to the type.

See `create annotation` for details.

`create property <property-name> ...` Define a concrete property item for this link. See `create property` for details.

`create constraint <constraint-name> ...` Define a concrete constraint for this link. See `create constraint` for details.

`on target delete <action>` Valid values for `action` are: `restrict`, `DELETE SOURCE`, `allow`, and `deferred restrict`. The details of what `on target delete` options mean are described in [this section](#).

`reset on target delete` Reset the delete policy to either the inherited value or to the default `restrict`. The details of what `on target delete` options mean are described in [this section](#).

`create index on <index-expr>` Define a new `index` using `index-expr` for this link. See `create index` for details.

### 8.3.4.1.3 Examples

Define a new link `friends` on the `User` object type:

```
alter type User {
    create multi link friends -> User
};
```

Define a new *computed link* `special_group` on the `User` object type, which contains all the friends from the same town:

```
alter type User {
    create link special_group := (
        select __source__.friends
        filter .town = __source__.town
    )
};
```

Define a new abstract link orderable and a concrete link `interests` that extends it, inheriting its `weight` property:

```
create abstract link orderable {
    create property weight -> std::int64
};

alter type User {
    create multi link interests extending orderable -> Interest
};
```

### 8.3.4.2 Alter link

`eql-statement`

`eql-haswith`

Change the definition of a *link*.

```
[ with <with-item> [, ...] ]
{create|alter} type <TypeName> "{"
[ ... ]
alter link <name>
[ "{" ] <subcommand>; [...] [ "}" ];
[ ... ]
}"

[ with <with-item> [, ...] ]
alter abstract link [<module>::]<name>
[ "{" ] <subcommand>; [...] [ "}" ];

# where <subcommand> is one of

set default := <expression>
reset default
set readonly := {true | false}
reset readonly
rename to <newname>
extending ...
set required
set optional
reset optionality
set single
set multi
reset cardinality
set type <typename> [using (<conversion-expr>)]
reset type
using (<computed-expr>)
create annotation <annotation-name> := <value>
alter annotation <annotation-name> := <value>
drop annotation <annotation-name>
create property <property-name> ...
alter property <property-name> ...
```

(continues on next page)

(continued from previous page)

```
drop property <property-name> ...
create constraint <constraint-name> ...
alter constraint <constraint-name> ...
drop constraint <constraint-name> ...
on target delete <action>
create index on <index-expr>
drop index on <index-expr>
```

### 8.3.4.2.1 Description

The combinations of `create type ... alter link` and `alter type ... alter link` change the definition of a concrete link for a given object type.

The command `alter abstract link` changes the definition of an abstract link item. *name* must be the identity of an existing abstract link, optionally qualified with a module name.

### 8.3.4.2.2 Parameters

The following subcommands are allowed in the `alter link` block:

**rename to** <*newname*> Change the name of the link item to *newname*. All concrete links inheriting from this links are also renamed.

**extending** ... Alter the link parent list. The full syntax of this subcommand is:

```
extending <name> [, ...]
[ first | last | before <parent> | after <parent> ]
```

This subcommand makes the link a child of the specified list of parent links. The requirements for the parent-child relationship are the same as when creating a link.

It is possible to specify the position in the parent list using the following optional keywords:

- **first** – insert parent(s) at the beginning of the parent list,
- **last** – insert parent(s) at the end of the parent list,
- **before** <parent> – insert parent(s) before an existing *parent*,
- **after** <parent> – insert parent(s) after an existing *parent*.

**set required** Make the link *required*.

**set optional** Make the link no longer *required* (i.e. make it *optional*).

**reset optionality** Reset the optionality of the link to the default value (*optional*), or, if the link is inherited, to the value inherited from links in supertypes.

**set single** Change the link set's maximum cardinality to *one*. Only valid for concrete links.

**set multi** Remove the upper limit on the link set's cardinality. Only valid for concrete links.

**reset cardinality** Reset the link set's maximum cardinality to the default value (**single**), or to the value inherited from the link's supertypes.

**set type** <*typename*> [**using** (<*conversion-expr*>)] Change the type of the link to the specified <*typename*>. The optional *using* clause specifies a conversion expression that computes the new link value from the old. The

conversion expression must return a singleton set and is evaluated on each element of `multi` links. A `using` clause must be provided if there is no implicit or assignment cast from old to new type.

**reset type** Reset the type of the link to be strictly the inherited type. This only has an effect on links that have been *overloaded* in order to change their inherited type. It is an error to `reset type` on a link that is not inherited.

**using (<computed-expr>)** Change the expression of a *computed link*. Only valid for concrete links.

**alter annotation <annotation-name>;** Alter link annotation `<annotation-name>`. See [alter annotation](#) for details.

**drop annotation <annotation-name>;** Remove link item's annotation `<annotation-name>`. See [drop annotation](#) for details.

**alter property <property-name> ...** Alter the definition of a property item for this link. See [alter property](#) for details.

**drop property <property-name>;** Remove a property item from this link. See [drop property](#) for details.

**alter constraint <constraint-name> ...** Alter the definition of a constraint for this link. See [alter constraint](#) for details.

**drop constraint <constraint-name>;** Remove a constraint from this link. See [drop constraint](#) for details.

**drop index on <index-expr>** Remove an `index` defined on `index-expr` from this link. See [drop index](#) for details.

**reset default** Remove the default value from this link, or reset it to the value inherited from a supertype, if the link is inherited.

**reset readonly** Set link writability to the default value (writable), or, if the link is inherited, to the value inherited from links in supertypes.

All the subcommands allowed in the `create link` block are also valid subcommands for `alter link` block.

#### 8.3.4.2.3 Examples

On the object type `User`, set the `title` annotation of its `friends` link to "Friends":

```
alter type User {
    alter link friends create annotation title := "Friends";
};
```

Rename the abstract link `orderable` to `sorted`:

```
alter abstract link orderable rename to sorted;
```

Redefine the *computed link* `special_group` to be those who have some shared interests:

```
alter type User {
    create link special_group := (
        select __source__.friends
        # at least one of the friend's interests
        # must match the user's
        filter .interests IN __source__.interests
    );
};
```

### 8.3.4.3 Drop link

**eql-statement**

**eql-haswith**

Remove the specified link from the schema.

```
[ with <with-item> [, ...] ]
alter type <TypeName> "{"
  [ ... ]
  drop link <name>
  [ ... ]
}"

[ with <with-item> [, ...] ]
drop abstract link [<module>]::<name>
```

#### 8.3.4.3.1 Description

The combination of `alter type` and `drop link` removes the specified link from its containing object type. All links that inherit from this link are also removed.

The command `drop abstract link` removes an existing link item from the database schema. All subordinate schema items defined on this link, such as link properties and constraints, are removed as well.

#### 8.3.4.3.2 Examples

Remove link `friends` from object type `User`:

```
alter type User drop link friends;
```

Drop abstract link `orderable`:

```
drop abstract link orderable;
```

See also
<a href="#">Schema &gt; Links</a>
<a href="#">SDL &gt; Links</a>
<a href="#">Introspection &gt; Object types</a>

## 8.3.5 Properties

This section describes the DDL commands pertaining to *properties*.

### 8.3.5.1 Create property

**eql-statement**

**eql-haswith**

*Define* a new property.

```
[ with <with-item> [, ...] ]
{create|alter} {type|link} <SourceName> "{"
  [ ... ]
  create [{required | optional}]
    [{single | multi}]
      property <name>
        [ extending <base> [, ...] ] -> <type>
        [ "{" <subcommand>; [...] "}" ] ;
  [ ... ]
}"

# Computed property form:

[ with <with-item> [, ...] ]
{create|alter} {type|link} <SourceName> "{"
  [ ... ]
  create [{required | optional}]
    [{single | multi}]
      property <name> := <expression>;
  [ ... ]
"

# Abstract property form:

[ with <with-item> [, ...] ]
create abstract property [<module>::]<name> [extending <base> [, ...]]
[ "{" <subcommand>; [...] "}" ]

# where <subcommand> is one of

  set default := <expression>
  set readonly := {true | false}
  create annotation <annotation-name> := <value>
  create constraint <constraint-name> ...
```

#### 8.3.5.1.1 Description

The combination `{create|alter} {type|link} ... create property` defines a new concrete property for a given object type or link.

There are three forms of `create property`, as shown in the syntax synopsis above. The first form is the canonical definition form, the second form is a syntax shorthand for defining a *computed property*, and the third is a form to define an abstract property item. The abstract form allows creating the property in the specified `<module>`. Concrete property forms are always created in the same module as the containing object or property.

### 8.3.5.1.2 Parameters

Most sub-commands and options of this command are identical to the [SDL property declaration](#). The following sub-commands are allowed in the `create` property block:

**set default := <expression>** Specifies the default value for the property as an EdgeQL expression. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

**set readonly := {true | false}** Specifies whether the property is considered *read-only*. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

**create annotation <annotation-name> := <value>** Set property <annotation-name> to <value>.

See [create annotation](#) for details.

**create constraint** Define a concrete constraint on the property. See [create constraint](#) for details.

### 8.3.5.1.3 Examples

Define a new link address on the User object type:

```
alter type User {
    create property address -> str
};
```

Define a new *computed property* `number_of_connections` on the User object type counting the number of interests:

```
alter type User {
    create property number_of_connections :=
        count(.interests)
};
```

Define a new abstract link orderable with `weight` property:

```
create abstract link orderable {
    create property weight -> std::int64
};
```

### 8.3.5.2 Alter property

**eql-statement**

**eql-haswith**

Change the definition of a *property*.

```
[ with <with-item> [, ...] ]
{create | alter} {type | link} <source> "{"
    [ ... ]
    alter property <name>
        [ "{" ] <subcommand>; [...] [ "}" ];
        [ ... ]
    "}"
```

(continues on next page)

(continued from previous page)

```
[ with <with-item> [, ...] ]
alter abstract property [<module>::]<name>
[ "{" ] <subcommand>; [...] [ "}" ];

# where <subcommand> is one of

set default := <expression>
reset default
set readonly := {true | false}
reset readonly
rename to <newname>
extending ...
set required
set optional
reset optionalilily
set single
set multi
reset cardinality
set type <typename> [using (<conversion-expr>)]
reset type
using (<computed-expr>)
create annotation <annotation-name> := <value>
alter annotation <annotation-name> := <value>
drop annotation <annotation-name>
create constraint <constraint-name> ...
alter constraint <constraint-name> ...
drop constraint <constraint-name> ...
```

### 8.3.5.2.1 Description

The combination {create|alter} {type|link} ... create property defines a new concrete property for a given object type or link.

The command alter abstract property changes the definition of an abstract property item.

### 8.3.5.2.2 Parameters

**<source>** The name of an object type or link on which the property is defined. May be optionally qualified with module.

**<name>** The unqualified name of the property to modify.

**<module>** Optional name of the module to create or alter the abstract property in. If not specified, the current module is used.

The following subcommands are allowed in the alter link block:

**rename to <newname>** Change the name of the property to <newname>. All concrete properties inheriting from this property are also renamed.

**extending ...** Alter the property parent list. The full syntax of this subcommand is:

```
extending <name> [, ...]
  [ first | last | before <parent> | after <parent> ]
```

This subcommand makes the property a child of the specified list of parent property items. The requirements for the parent-child relationship are the same as when creating a property.

It is possible to specify the position in the parent list using the following optional keywords:

- **first** – insert parent(s) at the beginning of the parent list,
- **last** – insert parent(s) at the end of the parent list,
- **before** <parent> – insert parent(s) before an existing *parent*,
- **after** <parent> – insert parent(s) after an existing *parent*.

**set required** Make the property *required*.

**set optional** Make the property no longer *required* (i.e. make it *optional*).

**reset optionality** Reset the optionality of the property to the default value (*optional*), or, if the property is inherited, to the value inherited from properties in supertypes.

**set single** Change the maximum cardinality of the property set to *one*. Only valid for concrete properties.

**set multi** Change the maximum cardinality of the property set to *greater than one*. Only valid for concrete properties.

**reset cardinality** Reset the maximum cardinality of the property to the default value (*single*), or, if the property is inherited, to the value inherited from properties in supertypes.

**set type** <typename> [**using** (<conversion-expr>)] Change the type of the property to the specified <typename>. The optional **using** clause specifies a conversion expression that computes the new property value from the old. The conversion expression must return a singleton set and is evaluated on each element of *multi* properties. A **using** clause must be provided if there is no implicit or assignment cast from old to new type.

**reset type** Reset the type of the property to the type inherited from properties of the same name in supertypes. It is an error to **reset type** on a property that is not inherited.

**using** (<computed-expr>) Change the expression of a *computed property*. Only valid for concrete properties.

**alter annotation** <annotation-name>; Alter property annotation <annotation-name>. See [alter annotation](#) for details.

**drop annotation** <annotation-name>; Remove property annotation <annotation-name>. See [drop annotation](#) for details.

**alter constraint** <constraint-name> ... Alter the definition of a constraint for this property. See [alter constraint](#) for details.

**drop constraint** <constraint-name>; Remove a constraint from this property. See [drop constraint](#) for details.

**reset default** Remove the default value from this property, or reset it to the value inherited from a supertype, if the property is inherited.

**reset readonly** Set property writability to the default value (writable), or, if the property is inherited, to the value inherited from properties in supertypes.

All the subcommands allowed in the **create property** block are also valid subcommands for **alter property** block.

### 8.3.5.2.3 Examples

Set the `title` annotation of property `address` of object type `User` to "Home address":

```
alter type User {
    alter property address
        create annotation title := "Home address";
};
```

Add a maximum-length constraint to property `address` of object type `User`:

```
alter type User {
    alter property address {
        create constraint max_len_value(500);
    };
};
```

Rename the property `weight` of link `orderable` to `sort_by`:

```
alter abstract link orderable {
    alter property weight rename to sort_by;
};
```

Redefine the `computed property` `number_of_connections` to be the number of friends:

```
alter type User {
    alter property number_of_connections using (
        count(.friends)
    );
};
```

### 8.3.5.3 Drop property

**eql-statement**

**eql-haswith**

Remove a `property` from the schema.

```
[ with <with-item> [, ...] ]
{create|alter} type <TypeName> "{"
    [ ... ]
    drop link <name>
    [ ... ]
}"

[ with <with-item> [, ...] ]
drop abstract property <name> ;
```

### 8.3.5.3.1 Description

The combination `alter {type|link} drop property` removes the specified property from its containing object type or link. All properties that inherit from this property are also removed.

The command `drop abstract property` removes the specified abstract property item from the schema.

### 8.3.5.3.2 Example

Remove property `address` from type `User`:

```
alter type User {
    drop property address;
};
```

See also
<a href="#">Schema &gt; Properties</a>
<a href="#">SDL &gt; Properties</a>
<a href="#">Introspection &gt; Object types</a>

## 8.3.6 Aliases

This section describes the DDL commands pertaining to *expression aliases*.

### 8.3.6.1 Create alias

`eql-statement`

`eql-haswith`

*Define* a new expression alias in the schema.

```
[ with <with-item> [ , ... ] ]
create alias <alias-name> := <alias-expr> ;

[ with <with-item> [ , ... ] ]
create alias <alias-name> "{"
    using <alias-expr>;
    [ create annotation <attr-name> := <attr-value>; ... ]
"}" ;

# where <with-item> is:

[ <module-alias> := ] module <module-name>
```

### 8.3.6.1.1 Description

The command `create alias` defines a new expression alias in the schema. The schema-level expression aliases are functionally equivalent to expression aliases defined in a statement `with block`, but are available to all queries using the schema and can be introspected.

If `name` is qualified with a module name, then the alias is created in that module, otherwise it is created in the current module. The alias name must be distinct from that of any existing schema item in the module.

### 8.3.6.1.2 Parameters

Most sub-commands and options of this command are identical to the [SDL alias declaration](#), with some additional features listed below:

[ `<module-alias> :=`  ] `module <module-name>` An optional list of module alias declarations to be used in the alias definition.

`create annotation <annotation-name> := <value>;` An optional list of annotation values for the alias. See [create annotation](#) for details.

### 8.3.6.1.3 Example

Create a new alias:

```
create alias Superusers := (
    select User filter User.groups.name = 'Superusers'
);
```

## 8.3.6.2 Drop alias

**eql-statement**

**eql-haswith**

Remove an expression alias from the schema.

```
[ with <with-item> [, ...] 
drop alias <alias-name> ;
```

### 8.3.6.2.1 Description

The command `drop alias` removes an expression alias from the schema.

### 8.3.6.2.2 Parameters

**alias-name** The name (optionally qualified with a module name) of an existing expression alias.

### 8.3.6.2.3 Example

Remove an alias:

```
drop alias SuperUsers;
```

See also
<a href="#">Schema &gt; Aliases</a>
<a href="#">SDL &gt; Aliases</a>
<a href="#">Cheatsheets &gt; Aliases</a>

## 8.3.7 Indexes

This section describes the DDL commands pertaining to *indexes*.

### 8.3.7.1 Create index

#### eql-statement

*Define* an new index for a given object type or link.

```
create index on ( <index-expr> )
[ except ( <except-expr> ) ]
[ "{" <subcommand>; [...] "}" ] ;

# where <subcommand> is one of

create annotation <annotation-name> := <value>
```

#### 8.3.7.1.1 Description

The command `create index` constructs a new index for a given object type or link using *index-expr*.

#### 8.3.7.1.2 Parameters

Most sub-commands and options of this command are identical to the [SDL index declaration](#). There's only one sub-command that is allowed in the `create index` block:

`create annotation <annotation-name> := <value>` Set object type *<annotation-name>* to *<value>*.

See `create annotation` for details.

### 8.3.7.1.3 Example

Create an object type `User` with an indexed `name` property:

```
create type User {
    create property name -> str {
        set default := '';
    };

    create index on (.name);
};
```

### 8.3.7.2 Alter index

#### eql-statement

Alter the definition of an *index*.

```
alter index on ( <index-expr> ) [ except ( <except-expr> ) ]
[ "{" <subcommand>; [...] "}" ] ;

# where <subcommand> is one of

create annotation <annotation-name> := <value>
alter annotation <annotation-name> := <value>
drop annotation <annotation-name>
```

#### 8.3.7.2.1 Description

The command `alter index` is used to change the *annotations* of an index. The *index-expr* is used to identify the index to be altered.

#### 8.3.7.2.2 Parameters

**on ( <index-expr> )** The specific expression for which the index is made. Note also that *<index-expr>* itself has to be parenthesized.

The following subcommands are allowed in the `alter index` block:

**create annotation <annotation-name> := <value>** Set index *<annotation-name>* to *<value>*. See `create annotation` for details.

**alter annotation <annotation-name>;** Alter index *<annotation-name>*. See `alter annotation` for details.

**drop annotation <annotation-name>;** Remove constraint *<annotation-name>*. See `drop annotation` for details.

### 8.3.7.2.3 Example

Add an annotation to the index on the `name` property of object type `User`:

```
alter type User {
    alter index on (.name) {
        create annotation title := "User name index";
    };
};
```

### 8.3.7.3 Drop index

#### eql-statement

Remove an index from a given schema item.

```
drop index on ( <index-expr> ) [ except ( <except-expr> ) ];
```

#### 8.3.7.3.1 Description

The command `drop index` removes an index from a schema item.

**on ( <index-expr> )** The specific expression for which the index was made.

This statement can only be used as a subdefinition in another DDL statement.

#### 8.3.7.3.2 Example

Drop the `name` index from the `User` object type:

```
alter type User {
    drop index on (.name);
};
```

See also
<a href="#">Schema &gt; Indexes</a>
<a href="#">SDL &gt; Indexes</a>
<a href="#">Introspection &gt; Indexes</a>

## 8.3.8 Constraints

This section describes the DDL commands pertaining to *constraints*.

### 8.3.8.1 Create abstract constraint

**eql-statement**

**eql-haswith**

*Define* a new abstract constraint.

```
[ with [ <module-alias> := ] module <module-name> ]
create abstract constraint <name> [ ( [<argspec>] [, ...] ) ]
[ on ( <subject-expr> ) ]
[ extending <base> [, ...] ]
"{" <subcommand>; [...] "}" ;

# where <argspec> is:
[ <argname>: ] <argtype>

# where <subcommand> is one of

using <constr-expression>
set errmessage := <error-message>
create annotation <annotation-name> := <value>
```

#### 8.3.8.1.1 Description

The command `create abstract constraint` defines a new abstract constraint.

If *name* is qualified with a module name, then the constraint is created in that module, otherwise it is created in the current module. The constraint name must be distinct from that of any existing schema item in the module.

#### 8.3.8.1.2 Parameters

Most sub-commands and options of this command are identical to the [SDL constraint declaration](#), with some additional features listed below:

**[ <module-alias> := ] module <module-name>** An optional list of module alias declarations to be used in the migration definition. When *module-alias* is not specified, *module-name* becomes the effective current module and is used to resolve all unqualified names.

**set errmessage := <error\_message>** An optional string literal defining the error message template that is raised when the constraint is violated. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

**create annotation <annotation-name> := <value>;** Set constraint <annotation-name> to <value>.

See [create annotation](#) for details.

### 8.3.8.1.3 Example

Create an abstract constraint “uppercase” which checks if the subject is a string in upper case.

```
create abstract constraint uppercase {
    create annotation title := "Upper case constraint";
    using (str_upper(__subject__) = __subject__);
    set errmessage := "{__subject__} is not in upper case";
};
```

### 8.3.8.2 Alter abstract constraint

**eql-statement**

**eql-haswith**

Alter the definition of an *abstract constraint*.

```
[ with [ <module-alias> := ] module <module-name> ]
alter abstract constraint <name>
"{" <subcommand>; [...] "}";
# where <subcommand> is one of

rename to <newname>
using <constr-expression>
set errmessage := <error-message>
reset errmessage
create annotation <annotation-name> := <value>
alter annotation <annotation-name> := <value>
drop annotation <annotation-name>
```

#### 8.3.8.2.1 Description

The command `alter abstract constraint` changes the definition of an abstract constraint item. *name* must be a name of an existing abstract constraint, optionally qualified with a module name.

#### 8.3.8.2.2 Parameters

[ <module-alias> := ] **module** <module-name> An optional list of module alias declarations to be used in the migration definition. When *module-alias* is not specified, *module-name* becomes the effective current module and is used to resolve all unqualified names.

**<name>** The name (optionally module-qualified) of the constraint to alter.

The following subcommands are allowed in the `alter abstract constraint` block:

**rename to <newname>** Change the name of the constraint to *newname*. All concrete constraints inheriting from this constraint are also renamed.

**alter annotation <annotation-name>;** Alter constraint <annotation-name>. See `alter annotation` for details.

**drop annotation <annotation-name>;** Remove constraint <annotation-name>. See [drop annotation](#) for details.

**reset errmessage;** Remove the error message from this abstract constraint. The error message specified in the base abstract constraint will be used instead.

All the subcommands allowed in a `create abstract constraint` block are also valid subcommands for an `alter abstract constraint` block.

### 8.3.8.2.3 Example

Rename the abstract constraint “uppercase” to “upper\_case”:

```
alter abstract constraint uppercase rename to upper_case;
```

### 8.3.8.3 Drop abstract constraint

**eql-statement**

**eql-haswith**

Remove an [abstract constraint](#) from the schema.

```
[ with [ <module-alias> := ] module <module-name> ]
drop abstract constraint <name> ;
```

#### 8.3.8.3.1 Description

The command `drop abstract constraint` removes an existing abstract constraint item from the database schema. If any schema items depending on this constraint exist, the operation is refused.

#### 8.3.8.3.2 Parameters

**[ <module-alias> := ] module <module-name>** An optional list of module alias declarations to be used in the migration definition. When *module-alias* is not specified, *module-name* becomes the effective current module and is used to resolve all unqualified names.

**<name>** The name (optionally module-qualified) of the constraint to remove.

#### 8.3.8.3.3 Example

Drop abstract constraint `upper_case`:

```
drop abstract constraint upper_case;
```

### 8.3.8.4 Create constraint

#### eql-statement

Define a concrete constraint on the specified schema item.

```
[ with [ <module-alias> := ] module <module-name> ]
create [ delegated ] constraint <name>
  [ ( [<argspec>] [, ...] ) ]
  [ on ( <subject-expr> ) ]
  [ except ( <except-expr> ) ]
"{" <subcommand>; [...] "}" ;

# where <argspec> is:

[ <argname>: ] <argvalue>

# where <subcommand> is one of

set errmessage := <error-message>
create annotation <annotation-name> := <value>
```

#### 8.3.8.4.1 Description

The command `create constraint` defines a new concrete constraint. It can only be used in the context of `create scalar type`, `alter scalar type`, `create property`, `alter property`, `create link`, or `alter link`. `name` must be a name (optionally module-qualified) of previously defined abstract constraint.

#### 8.3.8.4.2 Parameters

Most sub-commands and options of this command are identical to the [SDL constraint declaration](#), with some additional features listed below:

**[ <module-alias> := ] module <module-name>** An optional list of module alias declarations to be used in the migration definition. When `module-alias` is not specified, `module-name` becomes the effective current module and is used to resolve all unqualified names.

**set errmessage := <error\_message>** An optional string literal defining the error message template that is raised when the constraint is violated. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

**create annotation <annotation-name> := <value>;** An optional list of annotations for the constraint. See `create annotation` for details.

### 8.3.8.4.3 Example

Create a “score” property on the “User” type with a minimum value constraint:

```
alter type User create property score -> int64 {
    create constraint min_value(0)
};
```

Create a Vector with a maximum magnitude:

```
create type Vector {
    create required property x -> float64;
    create required property y -> float64;
    create constraint expression ON (
        __subject__.x^2 + __subject__.y^2 < 25
    );
}
```

### 8.3.8.5 Alter constraint

#### eql-statement

Alter the definition of a concrete constraint on the specified schema item.

```
[ with [ <module-alias> := ] module <module-name> [, ...] ]
alter constraint <name>
[ ( [<argspec>] [, ...] ) ]
[ on ( <subject-expr> ) ]
[ except ( <except-expr> ) ]
"{" <subcommand>; [ ... ] "}" ;

# -- or --

[ with [ <module-alias> := ] module <module-name> [, ...] ]
alter constraint <name>
[ ( [<argspec>] [, ...] ) ]
[ on ( <subject-expr> ) ]
<subcommand> ;

# where <subcommand> is one of:

set delegated
set not delegated
set errmessage := <error-message>
reset errmessage
create annotation <annotation-name> := <value>
alter annotation <annotation-name>
drop annotation <annotation-name>
```

### 8.3.8.5.1 Description

The command `alter constraint` changes the definition of a concrete constraint. As for most `alter` commands, both single- and multi-command forms are supported.

### 8.3.8.5.2 Parameters

[ `<module-alias> := ] module <module-name>` An optional list of module alias declarations to be used in the migration definition. When `module-alias` is not specified, `module-name` becomes the effective current module and is used to resolve all unqualified names.

`<name>` The name (optionally module-qualified) of the concrete constraint that is being altered.

`<argspec>` A list of constraint arguments as specified at the time of `create constraint`.

`on ( <subject-expr> )` A expression defining the *subject* of the constraint as specified at the time of `create constraint`.

The following subcommands are allowed in the `alter constraint` block:

`set delegated` Makes the constraint delegated.

`set not delegated` Makes the constraint non-delegated.

`rename to <newname>` Change the name of the constraint to `<newname>`.

`alter annotation <annotation-name>;` Alter constraint `<annotation-name>`. See [alter annotation](#) for details.

`drop annotation <annotation-name>;` Remove an *annotation*. See [drop annotation](#) for details.

`reset errmessage;` Remove the error message from this constraint. The error message specified in the abstract constraint will be used instead.

All the subcommands allowed in the `create constraint` block are also valid subcommands for `alter constraint` block.

### 8.3.8.5.3 Example

Change the error message on the minimum value constraint on the property “score” of the “User” type:

```
alter type User alter property score
    alter constraint min_value(0)
        set errmessage := 'Score cannot be negative';
```

### 8.3.8.6 Drop constraint

`eql-statement`

`eql-haswith`

Remove a concrete constraint from the specified schema item.

```
[ with [ <module-alias> := ] module <module-name> [, ...] ]
drop constraint <name>
[ ( [<argspec>] [, ...] ) ]
```

(continues on next page)

(continued from previous page)

```
[ on ( <subject-expr> ) ]
[ except ( <except-expr> ) ] ;
```

### 8.3.8.6.1 Description

The command `drop constraint` removes the specified constraint from its containing schema item.

### 8.3.8.6.2 Parameters

[ **<module-alias> :=** ] **module <module-name>** An optional list of module alias declarations to be used in the migration definition. When *module-alias* is not specified, *module-name* becomes the effective current module and is used to resolve all unqualified names.

**<name>** The name (optionally module-qualified) of the concrete constraint to remove.

**<argspec>** A list of constraint arguments as specified at the time of `create constraint`.

**on ( <subject-expr> )** A expression defining the *subject* of the constraint as specified at the time of `create constraint`.

### 8.3.8.6.3 Example

Remove constraint “min\_value” from the property “score” of the “User” type:

```
alter type User alter property score
    drop constraint min_value();
```

See also
<a href="#">Schema &gt; Constraints</a>
<a href="#">SDL &gt; Constraints</a>
<a href="#">Introspection &gt; Constraints</a>
<a href="#">Standard Library &gt; Constraints</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; Constraints</a>

## 8.3.9 Annotations

This section describes the DDL commands pertaining to *annotations*.

### 8.3.9.1 Create abstract annotation

#### eql-statement

*Define* a new annotation.

```
[ with <with-item> [, ...] ]
create abstract [ inheritable ] annotation <name>
[ "{"]
    create annotation <annotation-name> := <value> ;
```

(continues on next page)

(continued from previous page)

```
[...]
"}" ] ;
```

### 8.3.9.1.1 Description

The command `create abstract annotation` defines a new annotation for use in the current database.

If `name` is qualified with a module name, then the annotation is created in that module, otherwise it is created in the current module. The annotation name must be distinct from that of any existing schema item in the module.

The annotations are non-inheritable by default. That is, if a schema item has an annotation defined on it, the descendants of that schema item will not automatically inherit the annotation. Normal inheritance behavior can be turned on by declaring the annotation with the `inheritable` qualifier.

Most sub-commands and options of this command are identical to the [SDL annotation declaration](#). There's only one subcommand that is allowed in the `create annotation` block:

```
create annotation <annotation-name> := <value> Annotations can also have annotations. Set the
<annotation-name> of the enclosing annotation to a specific <value>. See create annotation for details.
```

### 8.3.9.1.2 Example

Declare an annotation `extrainfo`.

```
create abstract annotation extrainfo;
```

### 8.3.9.2 Alter abstract annotation

#### eql-statement

Change the definition of an [annotation](#).

```
alter abstract annotation <name>
[ "{" ] <subcommand>; [...] [ "}" ];
# where <subcommand> is one of
  rename to <newname>
  create annotation <annotation-name> := <value>
  alter annotation <annotation-name> := <value>
  drop annotation <annotation-name>
```

### 8.3.9.2.1 Description

`alter abstract annotation` changes the definition of an abstract annotation.

### 8.3.9.2.2 Parameters

**<name>** The name (optionally module-qualified) of the annotation to alter.

The following subcommands are allowed in the `alter abstract annotation` block:

**rename to <newname>** Change the name of the annotation to `<newname>`.

**alter annotation <annotation-name>;** Annotations can also have annotations. Change `<annotation-name>` to a specific `<value>`. See [alter annotation](#) for details.

**drop annotation <annotation-name>;** Annotations can also have annotations. Remove annotation `<annotation-name>`. See [drop annotation](#) for details.

All the subcommands allowed in the `create abstract annotation` block are also valid subcommands for `alter annotation` block.

### 8.3.9.2.3 Examples

Rename an annotation:

```
alter abstract annotation extrainfo
    rename to extra_info;
```

## 8.3.9.3 Drop abstract annotation

### eql-statement

Remove a [schema annotation](#).

```
[ with <with-item> [, ...] ]
drop abstract annotation <name> ;
```

### 8.3.9.3.1 Description

The command `drop abstract annotation` removes an existing schema annotation from the database schema. Note that the `inheritable` qualifier is not necessary in this statement.

### 8.3.9.3.2 Example

Drop the annotation `extra_info`:

```
drop abstract annotation extra_info;
```

### 8.3.9.4 Create annotation

#### eql-statement

Define an annotation value for a given schema item.

```
create annotation <annotation-name> := <value>
```

#### 8.3.9.4.1 Description

The command `create annotation` defines an annotation for a schema item.

`<annotation-name>` refers to the name of a defined annotation, and `<value>` must be a constant EdgeQL expression evaluating into a string.

This statement can only be used as a subcommand in another DDL statement.

#### 8.3.9.4.2 Example

Create an object type `User` and set its `title` annotation to "User type".

```
create type User {
    create annotation title := "User type";
};
```

### 8.3.9.5 Alter annotation

#### eql-statement

Alter an annotation value for a given schema item.

```
alter annotation <annotation-name> := <value>
```

#### 8.3.9.5.1 Description

The command `alter annotation` alters an annotation value on a schema item.

`<annotation-name>` refers to the name of a defined annotation, and `<value>` must be a constant EdgeQL expression evaluating into a string.

This statement can only be used as a subcommand in another DDL statement.

#### 8.3.9.5.2 Example

Alter an object type `User` and alter the value of its previously set `title` annotation to "User type".

```
alter type User {
    alter annotation title := "User type";
};
```

### 8.3.9.6 Drop annotation

#### eql-statement

Remove an annotation from a given schema item.

```
drop annotation <annotation-name> ;
```

#### 8.3.9.6.1 Description

The command `drop annotation` removes an annotation value from a schema item.

`<annotation_name>` refers to the name of a defined annotation. The annotation value does not have to exist on a schema item.

This statement can only be used as a subcommand in another DDL statement.

#### 8.3.9.6.2 Example

Drop the `title` annotation from the `User` object type:

```
alter type User {
    drop annotation title;
};
```

See also
<a href="#">Schema &gt; Annotations</a>
<a href="#">SDL &gt; Annotations</a>
<a href="#">Cheatsheets &gt; Annotations</a>
<a href="#">Introspection &gt; Object types</a>

## 8.3.10 Globals

This section describes the DDL commands pertaining to global variables.

### 8.3.10.1 Create global

#### eql-statement

#### eql-haswith

*Declare* a new global variable.

```
[ with <with-item> [, ...] ]
create [{required | optional}] [single]
    global <name> -> <type>
    [ "{" <subcommand>; [...] "}" ] ;

# Computed global variable form:

[ with <with-item> [, ...] ]
```

(continues on next page)

(continued from previous page)

```
create [{required | optional}] [{single | multi}]
  global <name> := <expression>;
# where <subcommand> is one of
  set default := <expression>
  create annotation <annotation-name> := <value>
```

### 8.3.10.1.1 Description

There two different forms of `global` declaration, as shown in the syntax synopsis above. The first form is for defining a `global` variable that can be `set` in a session. The second form is not directly set, but instead it is *computed* based on an expression, potentially deriving its value from other `global` variables.

### 8.3.10.1.2 Parameters

Most sub-commands and options of this command are identical to the [SDL global variable declaration](#). The following subcommands are allowed in the `create global` block:

`set default := <expression>` Specifies the default value for the global variable as an EdgeQL expression. The default value is used by the session if the value was not explicitly specified or by the `reset` command.

`create annotation <annotation-name> := <value>` Set global variable `<annotation-name>` to `<value>`.

See `create annotation` for details.

### 8.3.10.1.3 Examples

Define a new global property `current_user_id`:

```
create global current_user_id -> uuid;
```

Define a new *computed* global property `current_user` based on the previously defined `current_user_id`:

```
create global current_user := (
  select User filter .id = global current_user_id
);
```

## 8.3.10.2 Alter global

`eql-statement`

`eql-haswith`

Change the definition of a global variable.

```
[ with <with-item> [, ...] ]
alter global <name>
[ "{" <subcommand>; [...] "}" ] ;
```

(continues on next page)

(continued from previous page)

```
# where <subcommand> is one of

set default := <expression>
reset default
rename to <newname>
set required
set optional
reset optionality
set single
set multi
reset cardinality
set type <typename> reset to default
using (<computed-expr>)
create annotation <annotation-name> := <value>
alter annotation <annotation-name> := <value>
drop annotation <annotation-name>
```

### 8.3.10.2.1 Description

The command `alter global` changes the definition of a global variable.

### 8.3.10.2.2 Parameters

**<name>** The name of the global variable to modify.

The following subcommands are allowed in the `alter global` block:

**reset default** Remove the default value from this global variable.

**rename to <newname>** Change the name of the global variable to `<newname>`.

**set required** Make the global variable *required*.

**set optional** Make the global variable no longer *required* (i.e. make it *optional*).

**reset optionality** Reset the optionality of the global variable to the default value (*optional*).

**set single** Change the maximum cardinality of the global variable to *one*.

**set multi** Change the maximum cardinality of the global variable set to *greater than one*. Only valid for computed global variables.

**reset cardinality** Reset the maximum cardinality of the global variable to the default value (*single*), or, if the property is computed, to the value inferred from its expression.

**set type <typename> reset to default** Change the type of the global variable to the specified `<typename>`. The `reset to default` clause is mandatory and it specifies that the variable will be reset to its default value after this command.

**using (<computed-expr>)** Change the expression of a computed global variable. Only valid for computed variables.

**alter annotation <annotation-name>;** Alter global variable annotation `<annotation-name>`. See [alter annotation](#) for details.

**drop annotation <annotation-name>;** Remove global variable `<annotation-name>`. See [drop annotation](#) for details.

All the subcommands allowed in the `create global` block are also valid subcommands for `alter global` block.

### 8.3.10.2.3 Examples

Set the `description` annotation of global variable `current_user`:

```
alter global current_user
    create annotation description :=
        'Current User as specified by the global ID';
```

Make the `current_user_id` global variable `required`:

```
alter global current_user_id {
    set required;
    # A required global variable MUST have a default value.
    set default := <uuid>'00ea8eaa-02f9-11ed-a676-6bd11cc6c557';
}
```

### 8.3.10.3 Drop global

**eql-statement**

**eql-haswith**

Remove a global variable from the schema.

```
[ with <with-item> [, ...] ]
drop global <name> ;
```

#### 8.3.10.3.1 Description

The command `drop global` removes the specified global variable from the schema.

#### 8.3.10.3.2 Example

Remove the `current_user` global variable:

```
drop global current_user;
```

See also
<a href="#">Schema &gt; Globals</a>
<a href="#">SDL &gt; Globals</a>

### 8.3.11 Access Policies

This section describes the DDL commands pertaining to access policies.

#### 8.3.11.1 Create access policy

##### eql-statement

*Declare* a new object access policy.

```
[ with <with-item> [, ...] ]
{ create | alter } type <TypeName> "{"
[ ... ]
create access policy <name>
[ when (<condition>) ; ]
{ allow | deny } <action> [, <action> ... ; ]
[ using (<expr>) ; ]
[ create annotation <annotation-name> := <value> ; ]
}"

# where <action> is one of
all
select
insert
delete
update [{ read | write }]
```

```
[ with with-item [, ...] ]
{ create | alter } type TypeName "{"
[ ... ]
create access policy name
[ when (condition) ; ]
{ allow | deny } action [, action ... ; ]
[ using (expr) ; ]
[ create annotation annotation-name := value ; ]
[ "{"
[ set errmessage := value ; ]
"}" ; ]
}"

# where <action> is one of
all
select
insert
delete
update [{ read | write }]
```

### 8.3.11.1.1 Description

The combination `{create | alter} type ... create access policy` defines a new access policy for a given object type.

### 8.3.11.1.2 Parameters

Most sub-commands and options of this command are identical to the [SDL access policy declaration](#).

**<name>** The name of the access policy.

**when (<condition>)** Specifies which objects this policy applies to. The `<condition>` has to be a `bool` expression.

When omitted, it is assumed that this policy applies to all objects of a given type.

**allow** Indicates that qualifying objects should allow access under this policy.

**deny** Indicates that qualifying objects should *not* allow access under this policy. This flavor supersedes any `allow` policy and can be used to selectively deny access to a subset of objects that otherwise explicitly allows accessing them.

**all** Apply the policy to all actions. It is exactly equivalent to listing `select`, `insert`, `delete`, `update` actions explicitly.

**select** Apply the policy to all selection queries. Note that any object that cannot be selected, cannot be modified either. This makes `select` the most basic “visibility” policy.

**insert** Apply the policy to all inserted objects. If a newly inserted object would violate this policy, an error is produced instead.

**delete** Apply the policy to all objects about to be deleted. If an object does not allow access under this kind of policy, it is not going to be considered by any `delete` command.

Note that any object that cannot be selected, cannot be modified either.

**update read** Apply the policy to all objects selected for an update. If an object does not allow access under this kind of policy, it is not visible cannot be updated.

Note that any object that cannot be selected, cannot be modified either.

**update write** Apply the policy to all objects at the end of an update. If an updated object violates this policy, an error is produced instead.

Note that any object that cannot be selected, cannot be modified either.

**update** This is just a shorthand for `update read` and `update write`.

Note that any object that cannot be selected, cannot be modified either.

**using <expr>** Specifies what the policy is with respect to a given eligible (based on `when` clause) object. The `<expr>` has to be a `bool` expression. The specific meaning of this value also depends on whether this policy flavor is `allow` or `deny`.

When omitted, it is assumed that this policy applies to all eligible objects of a given type.

The following subcommands are allowed in the `create access policy` block:

**set errmessage := <value>** Set a custom error message of `<value>` that is displayed when this access policy prevents a write action.

**create annotation <annotation-name> := <value>** Set access policy annotation `<annotation-name>` to `<value>`.

See `create annotation` for details.

### 8.3.11.2 Alter access policy

#### eql-statement

*Declare* a new object access policy.

```
[ with <with-item> [, ...] ]
alter type <TypeName> "{

  [ ... ]

  alter access policy <name> "{

    [ when (<condition>) ; ]
    [ reset when ; ]
    { allow | deny } <action> [, <action> ... ; ]
    [ using (<expr>) ; ]
    [ reset expression ; ]
    [ create annotation <annotation-name> := <value> ; ]
    [ alter annotation <annotation-name> := <value> ; ]
    [ drop annotation <annotation-name>; ]

  }"
}

# where <action> is one of
all
select
insert
delete
update [{ read | write }]
```

```
[ with <with-item> [, ...] ]
alter type <TypeName> "{

  [ ... ]

  alter access policy <name> "{

    [ when (<condition>) ; ]
    [ reset when ; ]
    { allow | deny } <action> [, <action> ... ; ]
    [ using (<expr>) ; ]
    [ set errmessage := value ; ]
    [ reset expression ; ]
    [ create annotation <annotation-name> := <value> ; ]
    [ alter annotation <annotation-name> := <value> ; ]
    [ drop annotation <annotation-name>; ]

  }"
}

# where <action> is one of
all
select
insert
delete
update [{ read | write }]
```

### 8.3.11.2.1 Description

The combination `{create | alter} type ... create access policy` defines a new access policy for a given object type.

### 8.3.11.2.2 Parameters

The parameters describing the action policy are identical to the parameters used by `create action policy`. There are a handful of additional subcommands that are allowed in the `create access policy` block:

**reset when** Clear the `when (<condition>)` so that the policy applies to all objects of a given type. This is equivalent to `when (true)`.

**reset expression** Clear the `using (<condition>)` so that the policy always passes. This is equivalent to `using (true)`.

**alter annotation <annotation-name>;** Alter access policy annotation `<annotation-name>`. See [alter annotation](#) for details.

**drop annotation <annotation-name>;** Remove access policy annotation `<annotation-name>`. See [drop annotation](#) for details.

All the subcommands allowed in the `create access policy` block are also valid subcommands for `alter access policy` block.

### 8.3.11.3 Drop access policy

#### eql-statement

Remove an access policy from an object type.

```
[ with <with-item> [, ...] ]
alter type <TypeName> "{"
  [ ... ]
  drop access policy <name> ;
"}"
```

### 8.3.11.3.1 Description

The combination `alter type ... drop access policy` removes the specified access policy from a given object type.

See also
<a href="#">Schema &gt; Access policies</a>
<a href="#">SDL &gt; Access policies</a>

## 8.3.12 Functions

This section describes the DDL commands pertaining to *functions*.

### 8.3.12.1 Create function

**eql-statement**

**eql-haswith**

*Define* a new function.

```
[ with <with-item> [, ...] ]
create function <name> ([ <argspec> ] [, ...]) -> <returnspec>
using ( <expr> );

[ with <with-item> [, ...] ]
create function <name> ([ <argspec> ] [, ...]) -> <returnspec>
using <language> <functionbody> ;

[ with <with-item> [, ...] ]
create function <name> ([ <argspec> ] [, ...]) -> <returnspec>
"{" <subcommand> [, ...] "}" ;

# where <argspec> is:

[ <argkind> ] <argname>: [ <typequal> ] <argtype> [ = <default> ]

# <argkind> is:

[ { variadic | named only } ]

# <typequal> is:

[ { set of | optional } ]

# and <returnspec> is:

[ <typequal> ] <rettype>

# and <subcommand> is one of

set volatility := {'Immutable' | 'Stable' | 'Volatile'} ;
create annotation <annotation-name> := <value> ;
using ( <expr> ) ;
using <language> <functionbody> ;
```

### 8.3.12.1.1 Description

The command `create function` defines a new function. If `name` is qualified with a module name, then the function is created in that module, otherwise it is created in the current module.

The function name must be distinct from that of any existing function with the same argument types in the same module. Functions of different argument types can share a name, in which case the functions are called *overloaded functions*.

### 8.3.12.1.2 Parameters

Most sub-commands and options of this command are identical to the *SDL function declaration*, with some additional features listed below:

**set volatility := {`'Immutable'` | `'Stable'` | `'Volatile'`}** Function volatility determines how aggressively the compiler can optimize its invocations. Other than a slight syntactical difference this is the same as the corresponding SDL declaration.

**create annotation <annotation-name> := <value>** Set the function's <annotation-name> to <value>.

See `create annotation` for details.

### 8.3.12.1.3 Examples

Define a function returning the sum of its arguments:

```
create function mysum(a: int64, b: int64) -> int64
using (
    select a + b
);
```

The same, but using a variadic argument and an explicit language:

```
create function mysum(variadic argv: int64) -> int64
using edgeql $$ 
    select sum(array_unpack(argv))
$$;
```

Define a function using the block syntax:

```
create function mysum(a: int64, b: int64) -> int64 {
    using (
        select a + b
    );
    create annotation title := "My sum function.";
};
```

### 8.3.12.2 Alter function

**eql-statement**

**eql-haswith**

Change the definition of a function.

```
[ with <with-item> [, ...] ]
alter function <name> ([ <argspec> ] [, ...]) "{"
    <subcommand> [, ...]
}"

# where <argspec> is:

[ <argkind> ] <argname>: [ <typequal> ] <argtype> [= <default>]

# and <subcommand> is one of

set volatility := {'Immutable' | 'Stable' | 'Volatile'} ;
reset volatility ;
rename to <newname> ;
create annotation <annotation-name> := <value> ;
alter annotation <annotation-name> := <value> ;
drop annotation <annotation-name> ;
using ( <expr> ) ;
using <language> <functionbody> ;
```

#### 8.3.12.2.1 Description

The command `alter function` changes the definition of a function. The command allows to change annotations, the volatility level, and other attributes.

#### 8.3.12.2.2 Subcommands

The following subcommands are allowed in the `alter function` block in addition to the commands common to the `create function`:

**reset volatility** Remove explicitly specified volatility in favor of the volatility inferred from the function body.

**rename to** <newname> Change the name of the function to *newname*.

**alter annotation** <annotation-name>; Alter function <annotation-name>. See [alter annotation](#) for details.

**drop annotation** <annotation-name>; Remove function <annotation-name>. See [drop annotation](#) for details.

**reset errmessage;** Remove the error message from this abstract constraint. The error message specified in the base abstract constraint will be used instead.

### 8.3.12.2.3 Example

```
create function mysum(a: int64, b: int64) -> int64 {
    using (
        select a + b
    );
    create annotation title := "My sum function.";
};

alter function mysum(a: int64, b: int64) {
    set volatility := 'Immutable';
    DROP ANNOTATION title;
};

alter function mysum(a: int64, b: int64) {
    using (
        select (a + b) * 100
    );
};
```

### 8.3.12.3 Drop function

eql-statement

eql-haswith

Remove a function.

```
[ with <with-item> [, ...] ]
drop function <name> ([ <argspec> ] [, ... ]);

# where <argspec> is:

[ <argkind> ] <argname>: [ <typequal> ] <argtype> [= <default> ]
```

#### 8.3.12.3.1 Description

The command `drop function` removes the definition of an existing function. The argument types to the function must be specified, since there can be different functions with the same name.

#### 8.3.12.3.2 Parameters

**<name>** The name (optionally module-qualified) of an existing function.

**<argname>** The name of an argument used in the function definition.

**<argmode>** The mode of an argument: `set` or `optional` or `variadic`.

**<argtype>** The data type(s) of the function's arguments (optionally module-qualified), if any.

### 8.3.12.3.3 Example

Remove the `mysum` function:

```
drop function mysum(a: int64, b: int64);
```

See also
<a href="#">Schema &gt; Functions</a>
<a href="#">SDL &gt; Functions</a>
<a href="#">Reference &gt; Function calls</a>
<a href="#">Introspection &gt; Functions</a>
<a href="#">Cheatsheets &gt; Functions</a>
<a href="#">Tutorial &gt; Advanced EdgeQL &gt; User-Defined Functions</a>

## 8.3.13 Triggers

This section describes the DDL commands pertaining to *triggers*.

### 8.3.13.1 Create trigger

#### eql-statement

*Define* a new trigger.

```
{create | alter} type <type-name> "{"
    create trigger <name>
        after
        {insert | update | delete} [, ...]
        for {each | all}
        do <expr>
"}"
```

#### 8.3.13.1.1 Description

The command `create trigger` nested under `create type` or `alter type` defines a new trigger for a given object type.

The trigger name must be distinct from that of any existing trigger on the same type.

#### 8.3.13.1.2 Parameters

The options of this command are identical to the *SDL trigger declaration*.

### 8.3.13.1.3 Example

Declare a trigger that inserts a Log object for each new User object:

```
alter type User {
    create trigger log_insert after insert for each do (
        insert Log {
            action := 'insert',
            target_name := __new__.name
        }
    );
};
```

### 8.3.13.2 Drop trigger

#### eql-statement

Remove a trigger.

```
alter type <type-name> "{
    drop trigger <name>;
}"
```

### 8.3.13.2.1 Description

The command `drop trigger` inside an `alter type` block removes the definition of an existing trigger on the specified type.

### 8.3.13.2.2 Parameters

`<type-name>` The name (optionally module-qualified) of the type being triggered on.

`<name>` The name of the trigger.

### 8.3.13.2.3 Example

Remove the `log_insert` trigger on the `User` type:

```
alter type User {
    drop trigger log_insert;
};
```

#### See also

[Schema > Triggers](#)

[SDL > Triggers](#)

[Introspection > Triggers](#)

## 8.3.14 Mutation Rewrites

This section describes the DDL commands pertaining to *mutation rewrites*.

### 8.3.14.1 Create rewrite

#### eql-statement

*Define* a new mutation rewrite.

When creating a new property or link:

```
{create | alter} type <type-name> "{"
    create { property | link } <prop-or-link-name> -> <type> "{"
        create rewrite {insert | update} [, ...]
            using <expr>
        "}";
    "}";
}
```

When altering an existing property or link:

```
{create | alter} type <type-name> "{"
    alter { property | link } <prop-or-link-name> "{"
        create rewrite {insert | update} [, ...]
            using <expr>
        "}";
    "}";
}
```

#### 8.3.14.1.1 Description

The command `create rewrite` nested under `create type` or `alter type` and then under `create property/link` or `alter property/link` defines a new mutation rewrite for the given property or link on the given object.

#### 8.3.14.1.2 Parameters

`<type-name>` The name (optionally module-qualified) of the type containing the rewrite.

`<prop-or-link-name>` The name (optionally module-qualified) of the property or link being rewritten.

`insert | update [, ...]` The query type (or types) that are rewritten. Separate multiple values with commas to invoke the same rewrite for multiple types of queries.

#### 8.3.14.1.3 Examples

Declare two mutation rewrites on new properties: one that sets a `created` property when a new object is inserted and one that sets a `modified` property on each update:

```
alter type User {
    create property created -> datetime {
        create rewrite insert using (datetime_of_statement());
    };
}
```

(continues on next page)

(continued from previous page)

```
create property modified -> datetime {
    create rewrite update using (datetime_of_statement());
};

};
```

### 8.3.14.2 Drop rewrite

#### eql-statement

Remove a mutation rewrite.

```
alter type <type-name> "{
    alter property <prop-or-link-name> "{
        drop rewrite {insert | update} ;
    }" ;
}" ;
```

#### 8.3.14.2.1 Description

The command `drop rewrite` inside an `alter type` block and further inside an `alter property` block removes the definition of an existing mutation rewrite on the specified property or link of the specified type.

#### 8.3.14.2.2 Parameters

`<type-name>` The name (optionally module-qualified) of the type containing the rewrite.

`<prop-or-link-name>` The name (optionally module-qualified) of the property or link being rewritten.

`insert | update [, ...]` The query type (or types) that are rewritten. Separate multiple values with commas to invoke the same rewrite for multiple types of queries.

#### 8.3.14.2.3 Example

Remove the `insert` rewrite of the `created` property on the `User` type:

```
alter type User {
    alter property created {
        drop rewrite insert;
    };
};
```

See also
<a href="#">Schema &gt; Mutation rewrites</a>
<a href="#">SDL &gt; Mutation rewrites</a>
<a href="#">Introspection &gt; Mutation rewrites</a>

## 8.3.15 Extensions

This section describes the DDL commands pertaining to *extensions*.

### 8.3.15.1 Create extension

#### eql-statement

Enable a particular extension for the current schema.

```
create extension <ExtensionName> ";"
```

There's a *corresponding SDL declaration* for enabling an extension, which is the recommended way of doing this.

#### 8.3.15.1.1 Description

The command `create extension` enables the specified extension for the current database.

#### 8.3.15.1.2 Examples

Enable *GraphQL* extension for the current schema:

```
create extension graphql;
```

Enable *EdgeQL over HTTP* extension for the current database:

```
create extension edgeql_http;
```

### 8.3.15.2 drop extension

#### eql-statement

Disable an extension.

```
drop extension <ExtensionName> ";"
```

#### 8.3.15.2.1 Description

The command `drop extension` disables a currently active extension for the current database.

#### 8.3.15.2.2 Examples

Disable *GraphQL* extension for the current schema:

```
drop extension graphql;
```

Disable *EdgeQL over HTTP* extension for the current database:

```
drop extension edgeql_http;
```

## 8.3.16 Future Behavior

This section describes the DDL commands pertaining to *future*.

### 8.3.16.1 Create future

#### eql-statement

Enable a particular future behavior for the current schema.

```
create future <FutureBehavior> ";"
```

There's a *corresponding SDL declaration* for enabling a future behavior, which is the recommended way of doing this.

#### 8.3.16.1.1 Description

The command `create future` enables the specified future behavior for the current database.

#### 8.3.16.1.2 Examples

Enable simpler non-recursive access policy behavior *non-recursive access policy* for the current schema:

```
create future nonrecursive_access_policies;
```

## 8.3.16.2 drop future

#### eql-statement

Stop importing future behavior prior to the EdgeDB version in which it appears.

```
drop future <FutureBehavior> ";"
```

#### 8.3.16.2.1 Description

The command `drop future` disables a currently active future behavior for the current database. However, this is only possible for versions of EdgeDB when the behavior in question is not officially introduced. Once a particular behavior is introduced as the standard behavior in an EdgeDB release, it cannot be disabled. Running this command will simply denote that no special action is needed to enable it in this case.

#### 8.3.16.2.2 Examples

Disable simpler non-recursive access policy behavior *non-recursive access policy* for the current schema. This will make access policy restrictions apply to the expressions defining other access policies:

```
drop future nonrecursive_access_policies;
```

Once EdgeDB 3.0 is released there is no more need for enabling non-recursive access policy behavior anymore. So the above command will simply indicate that the database no longer does anything non-standard.

## 8.3.17 Migrations

This section describes the DDL commands pertaining to migrations.

---

**Note:** Like all DDL commands, `start migration` and other migration commands are considered low-level. Users are encouraged to use the built-in [migration tools](#) instead.

---

### 8.3.17.1 Start migration

#### eql-statement

Start a migration block.

```
start migration to "{"  
    <sdl-declaration> ;  
    [ ... ]  
"}" ;
```

#### 8.3.17.1.1 Parameters

`<sdl-declaration>` Complete schema defined with the declarative [EdgeDB schema definition language](#).

#### 8.3.17.1.2 Description

The command `start migration` defines a migration of the schema to a new state. The target schema state is described using [SDL](#) and describes the entire schema. This is important to remember when creating a migration to add a few more things to an existing schema as all the existing schema objects and the new ones must be included in the `start migration` command. Objects that aren't included in the command will be removed from the new schema (which may result in data loss).

This command also starts a transaction block if not inside a transaction already.

While inside a migration block, all issued EdgeQL statements are not executed immediately and are instead recorded to be part of the migration script. Aside from normal EdgeQL commands the following special migration commands are available:

- `describe current migration` – return a list of statements currently recorded as part of the migration;
- `populate migration` – auto-populate the migration with system-generated DDL statements to achieve the target schema state;
- `abort migration` – abort the migration block and discard the migration;
- `commit migration` – commit the migration by executing the migration script statements and recording the migration into the system migration log.

### 8.3.17.1.3 Examples

Create a new migration to a target schema specified by the EdgeDB Schema syntax:

```
start migration to {
    module default {
        type User {
            property username -> str;
        };
    };
};
```

### 8.3.17.2 create migration

#### eql-statement

Create a new migration using an explicit EdgeQL script.

```
create migration "["  
    <edgeql-statement> ;  
    [ ... ]  
"]" ;
```

#### 8.3.17.2.1 Parameters

**<edgeql-statement>** Any valid EdgeQL statement, except database, role, configure, migration, or transaction statements.

#### 8.3.17.2.2 Description

The command `create migration` executes all the nested EdgeQL commands and records the migration into the system migration log.

#### 8.3.17.2.3 Examples

Create a new migration to a target schema specified by the EdgeDB Schema syntax:

```
create migration {
    create type default::User {
        create property username -> str;
    };
};
```

### 8.3.17.3 Abort migration

#### eql-statement

Abort the current migration block and discard the migration.

```
abort migration ;
```

#### 8.3.17.3.1 Description

The command `abort migration` is used to abort a migration block started by `start migration`. Issuing `abort migration` outside of a migration block is an error.

#### 8.3.17.3.2 Examples

Start a migration block and then abort it:

```
start migration to {
    module default {
        type User;
    };
};

abort migration;
```

### 8.3.17.4 Populate migration

#### eql-statement

Populate the current migration with system-generated statements.

```
populate migration ;
```

#### 8.3.17.4.1 Description

The command `populate migration` is used within a migration block started by `start migration` to automatically fill the migration with system-generated statements to achieve the desired target schema state. If the system is unable to automatically find a satisfactory sequence of statements to perform the migration, an error is returned. Issuing `populate migration` outside of a migration block is also an error.

**Warning:** The statements generated by `populate migration` may drop schema objects, which may result in data loss. Make sure to inspect the generated migration using `describe current migration` before running `commit migration`!

### 8.3.17.4.2 Examples

Start a migration block and populate it with auto-generated statements.

```
start migration to {
    module default {
        type User;
    };
};

populate migration;
```

### 8.3.17.5 Describe current migration

#### eql-statement

Describe the migration in the current migration block.

```
describe current migration [ as {ddl | json} ];
```

### 8.3.17.5.1 Description

The command `describe current migration` generates a description of the migration in the current migration block in the specified output format:

**as ddl** Show a sequence of statements currently recorded as part of the migration using valid [DDL](#) syntax. The output will indicate if the current migration is fully defined, i.e. the recorded statements bring the schema to the state specified by `start migration`.

**as json** Provide a machine-readable description of the migration using the following JSON format:

```
{
    // Name of the parent migration
    "parent": "<parent-migration-name>",

    // Whether the confirmed DDL makes the migration complete,
    // i.e. there are no more statements to issue.
    "complete": {true|false},

    // List of confirmed migration statements
    "confirmed": [
        "<stmt text>",
        ...
    ],
    // The variants of the next statement
    // suggested by the system to advance
    // the migration script.
    "proposed": {
        "statements": [
            {
                "text": "<stmt text template>"
            }
        ],
    }
};
```

(continues on next page)

(continued from previous page)

```

"required-user-input": [
    {
        "placeholder": "<placeholder variable>",
        "prompt": "<statement prompt>",
    },
    ...
],
"confidence": (0..1), // confidence coefficient
"prompt": "<operation prompt>",
"prompt_id": "<prompt id>",
// Whether the operation is considered to be non-destructive.
"data_safe": {true|false}
}
}

```

Where:

**<stmt text>** Regular statement text.

**<stmt text template>** Statement text template with interpolation points using the \(name) syntax.

**<placeholder variable>** The name of an interpolation variable in the statement text template for which the user prompt is given.

**<statement prompt>** The text of a user prompt for an interpolation variable.

**<operation prompt>** Prompt for the proposed migration step.

**<prompt id>** An opaque string identifier for a particular operation prompt. The client should not repeat prompts with the same prompt id.

### 8.3.17.6 Commit migration

#### eql-statement

Commit the current migration to the database.

```
commit migration ;
```

#### 8.3.17.6.1 Description

The command `commit migration` executes all the commands defined by the current migration and records the migration as the most recent migration in the database.

Issuing `commit migration` outside of a migration block initiated by `start migration` is an error.

### 8.3.17.6.2 Example

Create and execute the current migration:

```
commit migration;
```

### 8.3.17.7 Reset schema to initial

#### eql-statement

Reset the database schema to its initial state.

```
reset schema to initial ;
```

**Warning:** This command will drop all entities and, as a consequence, all data. You won't want to use this statement on a production instance unless you want to lose all that instance's data.

## 8.3.17.8 Migration Rewrites

Migration rewrites allow you to change the migration history as long as your final schema matches the current database schema.

### 8.3.17.8.1 Start migration rewrite

Start a migration rewrite.

```
start migration rewrite ;
```

Once the migration rewrite is started, you can run any arbitrary DDL until you are ready to [commit](#) your new migration history. The most useful DDL in this context will be [create migration](#) statements, which will allow you to create a sequence of migrations that will become your new migration history.

### 8.3.17.8.2 Declare savepoint

Establish a new savepoint within the current migration rewrite.

```
declare savepoint <savepoint-name> ;
```

#### Parameters

**<savepoint-name>** The name which will be used to identify the new savepoint if you need to later release it or roll back to it.

### 8.3.17.8.3 Release savepoint

Destroys a savepoint previously defined in the current migration rewrite.

```
release savepoint <savepoint-name> ;
```

#### Parameters

<savepoint-name> The name of the savepoint to be released.

### 8.3.17.8.4 Rollback to savepoint

Rollback to the named savepoint.

```
rollback to savepoint <savepoint-name> ;
```

All changes made after the savepoint are discarded. The savepoint remains valid and can be rolled back to again later, if needed.

#### Parameters

<savepoint-name> The name of the savepoint to roll back to.

### 8.3.17.8.5 Rollback

Rollback the entire migration rewrite.

```
rollback ;
```

All updates made within the transaction are discarded.

### 8.3.17.8.6 Commit migration rewrite

Commit a migration rewrite.

```
commit migration rewrite ;
```

**edb-alt-title** Data Definition Language

EdgeQL includes a set of *data definition language* (DDL) commands that manipulate the database's schema. DDL is the low-level equivalent to *EdgeDB schema definition language*. You can execute DDL commands against your database, just like any other EdgeQL query.

```
edgedb> create type Person {
.....     create required property name -> str;
..... };
OK: CREATE TYPE
edgedb> create type Movie {
.....     create required property title -> str;
```

(continues on next page)

(continued from previous page)

```
.....      create required link director -> Person;
.....  };
OK: CREATE TYPE
```

In DDL, the *order* of commands is important. In the example above, you couldn't create `Movie` before `Person`, because `Movie` contains a link to `Person`.

Under the hood, all migrations are represented as DDL scripts: a sequence of imperative commands representing the migration. When you [create a migration](#) with the CLI, EdgeDB produces a DDL script.

### 8.3.18 Comparison to SDL

SDL is sort of like a 3D printer: you design the final shape and it puts it together for you. DDL is like building a house with traditional methods: to add a window, you first need a frame, to have a frame you need a wall, and so on.

DDL lets you make quick changes to your schema without creating migrations. But it can be dangerous too; some DDL commands can destroy user data permanently. In practice, we recommend most users stick with SDL until they get comfortable, then start experimenting with DDL.

## 8.4 Connection parameters

- *Instance parameters*
- *Priority levels*
- *Granular parameters*

The CLI and client libraries (collectively referred to as “clients” below) must connect to an EdgeDB instance to run queries or commands. There are several connection parameters, each of which can be specified in several ways.

### 8.4.1 Specifying an instance

There are several ways to uniquely identify an EdgeDB instance.

Parameter	CLI flag	Environment variable
Instance name	--instance/-I <name>	EDGEDB_INSTANCE
DSN	--dsn <dsn>	EDGEDB_DSN
Host and port	--host/-H <host> --port/-P <port>	EDGEDB_HOST and EDGEDB_PORT
Credentials file	--credentials-file <path>	EDGEDB_CREDENTIALS_FILE
<i>Project linking</i>	N/A	N/A

Let's dig into each of these a bit more.

**Instance name** All local instances (instances created on your local machine using the CLI) are associated with a name. This name is that's needed to connect; under the hood, the CLI stores the instance credentials (username, password, etc) on your file system in the EdgeDB [config directory](#). The CLI and client libraries look up these credentials to connect.

You can also assign names to remote instances using [edgedb instance link](#). The CLI will save the credentials locally, so you can connect to a remote instance using just its name, just like a local instance.

**DSN** DSNs (data source names, also referred to as “connection strings”) are a convenient and flexible way to specify connection information with a simple string. It takes the following form:

```
edgedb://USERNAME:PASSWORD@HOSTNAME:PORT/DATABASE  
# example  
edgedb://alice:pa$$w0rd@example.com:1234/my_db
```

All components of the DSN are optional; technically `edgedb://` is a valid DSN. The unspecified values will fall back to their defaults:

```
Host: "localhost"  
Port: 5656  
User: "edgedb"  
Password: null  
Database name: "edgedb"
```

DSNs also accept query parameters to support advanced use cases. Read the [DSN Specification](#) reference for details.

**Host and port** In general, we recommend using a fully-qualified DSN when connecting to the database. For convenience, it’s possible to individually specify a host and/or a port.

When not otherwise specified, the host defaults to `"localhost"` and the port defaults to `5656`.

**Credentials file** e.g. `/path/to/credentials.json`.

If you wish, you can store your credentials as a JSON file. Checking this file into version control could present a security risk and is not recommended.

```
{  
  "host": "localhost",  
  "port": 10702,  
  "user": "testuser",  
  "password": "testpassword",  
  "database": "edgedb",  
  "tls_cert_data": "-----BEGIN CERTIFICATE-----\nabcdef..."  
}
```

Relative paths are resolved relative to the current working directory.

**Project-linked instances** When you run `edgedb project init` in a given directory, EdgeDB creates an instance and “links” it to that directory. There’s nothing magical about this link; it’s just a bit of metadata that gets stored in the EdgeDB config directory. When you use the client libraries or run a CLI command inside a project-linked directory, the library/CLI can detect this, look up the linked instance’s credentials, and connect automatically.

For more information on how this works, check out the [release post](#) for `edgedb project`.

## 8.4.2 Priority levels

The section above describes the various ways of specifying an EdgeDB instance. There are also several ways to provide this configuration information to the client. From highest to lowest priority, you can pass them explicitly as parameters/flags (useful for debugging), use environment variables (recommended for production), or rely on `edgedb project` (recommended for development).

- 1. Explicit connection parameters.** For security reasons, hard-coding connection information or credentials in your codebase is not recommended, though it may be useful for debugging or testing purposes. As such, explicitly provided parameters are given the highest priority.

In the context of the client libraries, this means passing an option explicitly into the `connect` call. Here's how this looks using the JavaScript library:

```
import * as edgedb from "edgedb";

const pool = await edgedb.connect({
  instance: "my_instance"
});
```

In the context of the CLI, this means using the appropriate command-line flags:

```
$ edgedb --instance my_instance
EdgeDB 2.x
Type \help for help, \quit to quit.
edgedb>
```

- 2. Environment variables.**

This is the recommended mechanism for providing connection information to your EdgeDB client, especially in production or when running EdgeDB inside a container. All clients read the following variables from the environment:

- `EDGEDB_DSN`
- `EDGEDB_INSTANCE`
- `EDGEDB_CREDENTIALS_FILE`
- `EDGEDB_HOST / EDGEDB_PORT`

When one of these environment variables is defined, there's no need to pass any additional information to the client. The CLI and client libraries will be able to connect without any additional information. You can execute CLI commands without any additional flags, like so:

```
$ edgedb # no flags needed
EdgeDB 2.x
Type \help for help, \quit to quit.
edgedb>
```

Using the JavaScript client library:

```
import * as edgedb from "edgedb";

const pool = edgedb.connect();
pool.query(`select 2 + 2;`).then(result => {
  // do stuff
})
```

**Warning:** Ambiguity is not permitted. For instance, specifying both `EDGEDB_INSTANCE` and `EDGEDB_DSN` will result in an error. You *can* use `EDGEDB_HOST` and `EDGEDB_PORT` simultaneously.

### 3. Project-linked credentials

If you are using `edgedb project` (which we recommend!) and haven't otherwise specified any connection parameters, the CLI and client libraries will connect to the instance that's been linked to your project.

This makes it easy to get up and running with EdgeDB. Once you've run `edgedb project init`, the CLI and client libraries will be able to connect to your database without any explicit flags or parameters, as long as you're inside the project directory.

If no connection information can be detected using the above mechanisms, the connection fails.

**Warning:** Within a given priority level, you cannot specify multiple instances "instance selection parameters" simultaneously. For instance, specifying both `EDGEDB_INSTANCE` and `EDGEDB_DSN` environment variables will result in an error.

#### 8.4.3 Granular parameters

The [instance selection](#) section describes several mechanisms for providing a complete set of connection information in a single package. Occasionally—perhaps in development or for testing—it may be useful to override a particular *component* of this configuration.

The following "granular" parameters will override any value set by the instance-level configuration object.

Environment variable	CLI flag
<code>EDGEDB_DATABASE</code>	<code>--database/-d &lt;name&gt;</code>
<code>EDGEDB_USER</code>	<code>--user/-u &lt;user&gt;</code>
<code>EDGEDB_PASSWORD</code>	<code>--password &lt;pass&gt;</code>
<code>EDGEDB_TLS_CA_FILE</code>	<code>--tls-ca-file &lt;path&gt;</code>
<code>EDGEDB_CLIENT_TLS_SECURITY</code>	<code>--tls-security</code>
<code>EDGEDB_CLIENT_SECURITY</code>	N/A

**EDGEDB\_DATABASE** Each EdgeDB *instance* can contain multiple *databases*. When an instance is created, a default database named `edgedb` is created. Unless otherwise specified, all incoming connections connect to the `edgedb` database.

**EDGEDB\_USER/EDGEDB\_PASSWORD** These are the credentials of the database user account to connect to the EdgeDB instance.

**EDGEDB\_TLS\_CA\_FILE** TLS is required to connect to any EdgeDB instance. To do so, the client needs a reference to the root certificate of your instance's certificate chain. Typically this will be handled for you when you create a local instance or link a remote one.

If you're using a globally trusted CA like Let's Encrypt, the root certificate will almost certainly exist already in your system's global certificate pool. In this case, you won't need to specify this path; it will be discovered automatically by the client.

If you're self-issuing certificates, you must download the root certificate and provide a path to its location on the filesystem. Otherwise TLS will fail to connect.

**EDGEDB\_CLIENT\_TLS\_SECURITY** Sets the TLS security mode. Determines whether certificate and hostname verification is enabled. Possible values:

- "strict" (**default**) — certificates and hostnames will be verified
- "no\_host\_verification" — verify certificates but not hostnames
- "insecure" — client libraries will trust self-signed TLS certificates. useful for self-signed or custom certificates.

This setting defaults to "strict" unless a custom certificate is supplied, in which case it is set to "no\_host\_verification".

**EDGEDB\_CLIENT\_SECURITY** Provides some simple “security presets”.

Currently there is only one valid value: `insecure_dev_mode`. Setting `EDGEDB_CLIENT_SECURITY=insecure_dev_mode` disables all TLS security measures. Currently it is equivalent to setting `EDGEDB_CLIENT_TLS_SECURITY=insecure` but it may encompass additional configuration settings later. This is most commonly used when developing locally with Docker.

#### 8.4.3.1 Override behavior

When specified, the connection parameters (user, password, and database) will *override* the corresponding element of a DSN, credentials file, etc. For instance, consider the following environment variables:

```
EDGEDB_DSN=edgedb://olduser:oldpass@hostname.com:5656
EDGEDB_USER=newuser
EDGEDB_PASSWORD=newpass
```

In this scenario, `newuser` will override `olduser` and `newpass` will override `oldpass`. The client library will try to connect using this modified DSN: `edgedb://newuser:newpass@hostname.com:5656`.

#### 8.4.3.2 Overriding across priority levels

This override behavior only happens *same or lower priority level*. For instance:

- `EDGEDB_PASSWORD` **will** override the password specified in `EDGEDB_DSN`
- `EDGEDB_PASSWORD` **will be ignored** if a DSN is passed explicitly using the `--dsn` flag. Explicit parameters take precedence over environment variables. To override the password of an explicit DSN, you need to pass it explicitly as well:

```
$ edgedb --dsn edgedb://username:oldpass@hostname.com --password qwerty
# connects to edgedb://username:qwerty@hostname.com
```

- `EDGEDB_PASSWORD` **will** override the stored password associated with a project-linked instance. (This is unlikely to be desirable.)

## 8.5 Environment Variables

The behavior of EdgeDB can be configured with environment variables. The variables documented on this page are supported when using the `edgedb-server` tool and the official [Docker image](#).

## 8.5.1 Variants

Some environment variables (noted below) support \*\_FILE and \*\_ENV variants.

- The \*\_FILE variant expects its value to be a file name. The file's contents will be read and used as the value.
- The \*\_ENV variant expects its value to be the name of another environment variable. The value of the other environment variable is then used as the final value. This is convenient in deployment scenarios where relevant values are auto populated into fixed environment variables.

## 8.5.2 Supported variables

### 8.5.2.1 EDGEDB\_SERVER\_BOOTSTRAP\_COMMAND

Useful to fine-tune initial user and database creation, and other initial setup.

Maps directly to the edgedb-server flag --default-auth-method. The \*\_FILE and \*\_ENV variants are also supported.

### 8.5.2.2 EDGEDB\_SERVER\_DEFAULT\_AUTH\_METHOD

Optionally specifies the authentication method used by the server instance. Supported values are SCRAM (the default) and Trust. When set to Trust, the database will allow complete unauthenticated access for all who have access to the database port.

This is often useful when setting an admin password on an instance that lacks one.

Use at your own risk and only for development and testing.

The \*\_FILE and \*\_ENV variants are also supported.

### 8.5.2.3 EDGEDB\_SERVER\_TLS\_CERT\_MODE

Specifies what to do when the TLS certificate and key are either not specified or are missing. When set to require\_file, the TLS certificate and key must be specified in the EDGEDB\_SERVER\_TLS\_CERT and EDGEDB\_SERVER\_TLS\_KEY variables and both must exist. When set to generate\_self\_signed a new self-signed certificate and private key will be generated and placed in the path specified by EDGEDB\_SERVER\_TLS\_CERT and EDGEDB\_SERVER\_TLS\_KEY, if those are set, otherwise the generated certificate and key are stored as edbtlsCert.pem and edbprivkey.pem in EDGEDB\_SERVER\_DATADIR, or, if EDGEDB\_SERVER\_DATADIR is not set then they will be placed in /etc/ssl/edgedb.

The default is generate\_self\_signed when EDGEDB\_SERVER\_SECURITY=insecure\_dev\_mode. Otherwise the default is require\_file.

Maps directly to the edgedb-server flag --tls-cert-mode. The \*\_FILE and \*\_ENV variants are also supported.

#### 8.5.2.4 EDGEDB\_SERVER\_TLS\_CERT\_FILE/EDGEDB\_SERVER\_TLS\_KEY\_FILE

The TLS certificate and private key files, exclusive with EDGEDB\_SERVER\_TLS\_MODE=generate\_self\_signed. Maps directly to the edgedb-server flags --tls-cert-file and --tls-key-file.

#### 8.5.2.5 EDGEDB\_SERVER\_SECURITY

When set to insecure\_dev\_mode, sets EDGEDB\_SERVER\_DEFAULT\_AUTH\_METHOD to Trust (see above), and EDGEDB\_SERVER\_TLS\_MODE to generate\_self\_signed (unless an explicit TLS certificate is specified). Finally, if this option is set, the server will accept plaintext HTTP connections.

Use at your own risk and only for development and testing.

Maps directly to the edgedb-server flag --security.

#### 8.5.2.6 EDGEDB\_SERVER\_PORT

Specifies the network port on which EdgeDB will listen. The default is 5656.

Maps directly to the edgedb-server flag --port. The \*\_FILE and \*\_ENV variants are also supported.

#### 8.5.2.7 EDGEDB\_SERVER\_BIND\_ADDRESS

Specifies the network interface on which EdgeDB will listen.

Maps directly to the edgedb-server flag --bind-address. The \*\_FILE and \*\_ENV variants are also supported.

#### 8.5.2.8 EDGEDB\_SERVER\_DATADIR

Specifies a path where the database files are located. Defaults to /var/lib/edgedb/data. Cannot be specified at the same time with EDGEDB\_SERVER\_BACKEND\_DSN.

Maps directly to the edgedb-server flag --data-dir.

#### 8.5.2.9 EDGEDB\_SERVER\_BACKEND\_DSN

Specifies a PostgreSQL connection string in the [URI format](#). If set, the PostgreSQL cluster specified by the URI is used instead of the builtin PostgreSQL server. Cannot be specified at the same time with EDGEDB\_SERVER\_DATADIR.

Maps directly to the edgedb-server flag --backend-dsn. The \*\_FILE and \*\_ENV variants are also supported.

#### 8.5.2.10 EDGEDB\_SERVER\_RUNSTATE\_DIR

Specifies a path where EdgeDB will place its Unix socket and other transient files.

Maps directly to the edgedb-server flag --runstate-dir.

### 8.5.2.11 EDGEDB\_SERVER\_ADMIN\_UI

Set to `enabled` to enable the web-based administrative UI for the instance.

Maps directly to the `edgedb-server` flag `--admin-ui`.

## 8.6 Create a project

Projects are the most convenient way to develop applications with EdgeDB. This is the recommended approach.

To get started, navigate to the root directory of your codebase in a shell and run `edgedb project init`. You'll see something like this:

```
$ edgedb project init
No `edgedb.toml` found in this repo or above.
Do you want to initialize a new project? [Y/n]
> Y
Checking EdgeDB versions...
Specify the version of EdgeDB to use with this project [1-rc3]:
> # left blank for default
Specify the name of EdgeDB instance to use with this project:
> my_instance
Initializing EdgeDB instance...
Bootstrap complete. Server is up and running now.
Project initialized.
```

Let's unpack that.

1. First, it asks you to specify an EdgeDB version, defaulting to the most recent version you have installed. You can also specify a version you *don't* have installed, in which case it will be installed.
2. Then it asks you how you'd like to run EdgeDB: locally, in a Docker image, or in the cloud (coming soon!).
3. Then it asks for an instance name. If no instance currently exists with this name, it will be created (using the method you specified in #2).
4. Then it **links** the current directory to that instance. A “link” is represented as some metadata stored in EdgeDB’s *config directory*—feel free to peek inside to see how it’s stored.
5. Then it creates an `edgedb.toml` file, which marks this directory as an EdgeDB project.
6. Finally, it creates a `dbschema` directory and a `dbschema/default.esdl` schema file (if they don’t already exist).

## 8.6.1 FAQ

### 8.6.1.1 How does this help me?

Once you’ve initialized a project, your project directory is *linked* to a particular instance. That means, you can run CLI commands without connection flags. For instance, `edgedb -I my_instance migrate` becomes simply `edgedb migrate`. The CLI detects the existence of the `edgedb.toml` file, reads the current directory, and checks if it’s associated with an existing project. If it is, it looks up the credentials of the linked instance (they’re stored in a *standardized location*), uses that information to connect to the instance, and applies the command.

Similarly, all *client libraries* will use the same mechanism to auto-connect inside project directories, no hard-coded credentials required.

```

import edgedb from "edgedb";

- const pool = edgedb.createPool("my_instance");
+ const pool = edgedb.createPool();

```

#### 8.6.1.2 What do you mean *link*?

The “link” is just metaphor that makes projects easier to think about; in practice, it’s just a bit of metadata we store in the EdgeDB [config directory](#). When the CLI or client libraries try to connect to an instance, they read the current directory and cross-reference it against the list of initialized projects. If there’s a match, it reads the credentials of the project’s associated instance and auto-connects.

#### 8.6.1.3 How does this work in production?

It doesn’t. Projects are intended as a convenient development tool that make it easier to develop EdgeDB-backed applications locally. In production, you should provide instance credentials to your client library of choice using environment variables. See [Connection parameters](#) page for more information.

#### 8.6.1.4 What’s the `edgedb.toml` file?

The contents of this file aren’t terribly important; this most important thing is simply that the file exists, since it’s how the CLI knows that a directory is an instance-linked EdgeDB project.

But since we’re talking about it, `edgedb.toml` currently supports just one configuration setting: `server-version`. This lets you specify the EdgeDB version expected by this project. The value in the created `edgedb.toml` is determined by the EdgeDB version you selected during the setup process.

---

**Note:** If you’re not familiar with the TOML file format, it’s a very cool, minimal language for config files designed to be simpler than JSON or YAML—check out a short cheatsheet [here](#).

---

#### 8.6.1.5 How do I use `edgedb project` for existing codebases?

If you already have a project on your computer that uses EdgeDB, follow these steps to convert it into an EdgeDB project:

1. Navigate into the project directory (the one containing your `dbschema` directory).
2. Run `edgedb project init`.
3. When asked for an instance name, enter the name of the existing local instance you use for development.

This will create `edgedb.toml` and link your project directory to the instance. And you’re done! Try running some commands without connection flags. Feels good, right?

### 8.6.1.6 How does this make projects more portable?

Let's say you just cloned a full-stack application that uses EdgeDB. The project directory already contains an `edgedb.toml` file. What do you do?

Just run `edgedb project init` inside the directory! This is the beauty of `edgedb project`. You don't need to worry about creating an instance with a particular name, running on a particular port, creating users and passwords, specifying environment variables, or any of the other things that make setting up local databases hard. Running `edgedb project init` will install the necessary version of EdgeDB (if you don't already have it installed), create an instance, apply all unapplied migrations. Then you can start up the application and it should work out of the box.

### 8.6.1.7 How do I unlink a project?

If you want to remove the link between your project and its linked instance, run `edgedb project unlink` anywhere inside the project. This doesn't affect the instance, it continues running as before. After unlinking, can run `edgedb project init` inside project again to create or select a new instance.

```
$ edgedb project init
No `edgedb.toml` found in `~/path/to/my_project` or above.
Do you want to initialize a new project? [Y/n]
> Y
Specify the name of EdgeDB instance to use with this project
[default: my_project]:
> my_project
Checking EdgeDB versions...
Specify the version of EdgeDB to use with this project [default: 2.x]:
> 2.x
```

### 8.6.1.8 How do I use `edgedb project` with a non-local instance?

Sometimes you may want to work on an EdgeDB instance that is just not in your local development environment, like you may have a second workstation, or you want to test against a staging database shared by the team.

This is totally a valid case and EdgeDB fully supports it!

Before running `edgedb project init`, you just need to create a local link to the remote EdgeDB instance first:

```
$ edgedb instance link
Specify the host of the server [default: localhost]:
> 192.168.4.2
Specify the port of the server [default: 5656]:
> 10818
Specify the database user [default: edgedb]:
> edgedb
Specify the database name [default: edgedb]:
> edgedb
Unknown server certificate: SHA1:c38a7a90429b033dfaf7a81e08112a9d58d97286. Trust? [y/N]
> y
Password for 'edgedb':
Specify a new instance name for the remote server [default: 192_168_4_2_10818]:
> staging_db
Successfully linked to remote instance. To connect run:
  edgedb -I staging_db
```

Then you could run the normal `edgedb project init` and use `staging_db` as the instance name.

---

**Note:** When using an existing instance, make sure that the project source tree is in sync with the current migration revision of the instance. If the current revision in the database doesn't exist under `dbschema/migrations/`, it'll raise an error trying to migrate or create new migrations. In this case, you should update your local source tree to the revision that matches the current revision of the database.

---

## 8.7 DSN specification

DSNs (data source names) are a convenient and flexible way to specify connection information with a simple string. It takes the following form:

```
edgedb://USERNAME:PASSWORD@HOSTNAME:PORT/DATABASE
```

For instance, here is a typical DSN: `edgedb://alice:pa$$w0rd@example.com:1234/my_db`.

All components of the DSN are optional; in fact, `edgedb://` is a valid DSN. Any unspecified values will fall back to their defaults:

```
Host: "localhost"
Port: 5656
User: "edgedb"
Password: null
Database name: "edgedb"
```

### 8.7.1 Query parameters

DSNs also support query parameters (`?host=myhost.com`) to support advanced use cases. The value for a given parameter can be specified in three ways: directly (e.g. `?host=example.com`), by specifying an environment variable containing the value (`?host_env=HOST_VAR`), or by specifying a file containing the value (`?host_file=./hostname.txt`).

---

**Note:** For a breakdown of these configuration options, see [Reference > Connection Parameters](#).

---

Plain param	File param	Environment param
host	host_file	host_env
port	port_file	port_env
database	database_file	database_env
user	user_file	user_env
password	password_file	password_env
<code>tls_ca_file</code>	<code>tls_ca_file_file</code>	<code>tls_ca_file_env</code>
<code>tls_security</code>	<code>tls_security_file</code>	<code>tls_security_env</code>

**Plain params** These “plain” parameters can be used to provide values for options that can’t otherwise be reflected in the DSN, like TLS settings (described in more detail below).

You can’t specify the same setting both in the body of the DSN and in a query parameter. For instance, the DSN below is invalid, as the port is ambiguous.

```
edgedb://hostname.com:1234?port=5678
```

**File params** If you prefer to store sensitive credentials in local files, you can use file params to specify a path to a local UTF-8 encoded file. This file should contain a single line containing the relevant value.

```
edgedb://hostname.com:1234?user_file=./username.txt
```

```
# ./username.txt  
my_username
```

Relative params are resolved relative to the current working directory at the time of connection.

**Environment params** Environment params lets you specify a *pointer* to another environment variable. At runtime, the specified environment variable will be read. If it isn't set, an error will be thrown.

```
MY_PASSWORD=p@$$w0rd  
EDGEDB_DSN=edgedb://hostname.com:1234?password_env=MY_PASSWORD
```

## 8.8 Dump file format

This description uses the same *conventions* as the protocol description.

### 8.8.1 General Structure

Dump file is structure as follows:

1. Dump file format marker \xFF\xD8\x00\x00\xD8EDGEDB\x00DUMP\x00 (17 bytes)
2. Format version number \x00\x00\x00\x00\x00\x00\x00\x01 (8 bytes)
3. Header block
4. Any number of data blocks

### 8.8.2 General Dump Block

Both header and data blocks are formatted as follows:

```
struct DumpHeader {  
    int8          mtype;  
  
    // SHA1 hash sum of block data  
    byte         sha1sum[20];  
  
    // Length of message contents in bytes,  
    // including self.  
    int32        message_length;  
  
    // Block data. Should be treated in opaque way by a client.  
    byte         data[message_length];  
}
```

Upon receiving a protocol dump data message, the dump client should:

- Replace packet type:
  - @ (0x40) → H (0x48)
  - = (0x3d) → D (0x44)
- Prepend SHA1 checksum to the block
- Append the entire dump protocol message disregarding the first byte (the message type).

### 8.8.3 Header Block

Format:

```
struct DumpHeader {
    // Message type ('H')
    int8          mtype = 0x48;

    // SHA1 hash sum of block data
    byte         sha1sum[20];

    // Length of message contents in bytes,
    // including self.
    int32        message_length;

    // A set of message headers.
    Headers       headers;

    // Protocol version of the dump
    int16        major_ver;
    int16        minor_ver;

    // Schema data
    string        schema_ddl;

    // Type identifiers
    int32        num_types;
    TypeInfo      types[num_types];

    // Object descriptors
    int32        num_descriptors;
    ObjectDesc    descriptors[num_descriptors]
};

struct TypeInfo {
    string        type_name;
    string        type_class;
    byte         type_id[16];
}

struct ObjectDesc {
    byte         object_id[16];
    bytes        description;
}
```

(continues on next page)

(continued from previous page)

```

int16      num_dependencies;
byte       dependency_id[num_dependencies][16];
}

```

Known headers:

- 101 BLOCK\_TYPE – block type, always “T”
- 102 SERVER\_TIME – server time when dump is started as a floating point unix timestamp stringified
- 103 SERVER\_VERSION – full version of server as string
- 105 SERVER\_CATALOG\_VERSION – the catalog version of the server, as a 64-bit integer. The catalog version is an identifier that is incremented whenever a change is made to the database layout or standard library.

#### 8.8.4 Data Block

Format:

```

struct DumpBlock {
    // Message type ('=')
    int8          mtype = 0x3d;

    // Length of message contents in bytes,
    // including self.
    int32         message_length;

    // A set of message headers.
    Headers        headers;
}

```

Known headers:

- 101 BLOCK\_TYPE – block type, always “D”
- 110 BLOCK\_ID – block identifier (16 bytes of UUID)
- 111 BLOCK\_NUM – integer block index stringified
- 112 BLOCK\_DATA – the actual block data

### 8.9 Backend high-availability

High availability is a sophisticated and systematic challenge, especially for databases. To address the problem, EdgeDB server now supports selected highly-available backend Postgres clusters, namely in 2 categories:

- API-based HA
- Adaptive HA without API

When the backend HA feature is enabled in EdgeDB, EdgeDB server will try its best to detect and react to backend failovers, whether a proper API is available or not.

During backend failover, no frontend connections will be closed; instead, all incoming queries will fail with a retryable error until failover has completed successfully. If the query originates from a client that supports retrying transactions, these queries may be retried by the client until the backend connection is restored and the query can be properly resolved.

### 8.9.1 API-based HA

EdgeDB server accepts different types of backends by looking into the protocol of the `--backend-dsn` command-line parameter. EdgeDB supports the following DSN protocols currently:

- `stolon+consul+http://`
- `stolon+consul+https://`

When using these protocols, EdgeDB builds the actual DSN of the cluster's leader node by calling the corresponding API using credentials in the `--backend-dsn` and subscribes to that API for failover events. Once failover is detected, EdgeDB drops all backend connections and routes all new backend connections to the new leader node.

[Stolon](#) is an open-source cloud native PostgreSQL manager for PostgreSQL high availability. Currently, EdgeDB supports using a Stolon cluster as the backend in a Consul-based setup, where EdgeDB acts as a Stolon proxy. This way, you only need to manage Stolon sentinels and keepers, plus a Consul deployment. To use a Stolon cluster, run EdgeDB server with a DSN, like so:

```
$ edgedb-server \
  --backend-dsn stolon+consul+http://localhost:8500/my-cluster
```

EdgeDB will connect to the Consul HTTP service at `localhost:8500`, and subscribe to the updates of the cluster named `my-cluster`.

Using a regular `postgres://` DSN disables API-based HA.

### 8.9.2 Adaptive HA

EdgeDB also supports DNS-based generic HA backends. This may be a cloud database with multi-AZ failover or some custom HA Postgres cluster that keeps a DNS name always resolved to the leader node. Adaptive HA can be enabled with a switch in addition to a regular backend DSN:

```
$ edgedb-server \
  --backend-dsn postgres://xxx.rds.amazonaws.com \
  --enable-backend-adaptive-ha
```

Once enabled, EdgeDB server will keep track of unusual backend events like unexpected disconnects or Postgres shutdown notifications. When a threshold is reached, EdgeDB considers the backend to be in the “failover” state. It then drops all current backend connections and try to re-establish new connections with the same backend DSN. Because EdgeDB doesn't cache resolved DNS values, the new connections will be established with the new leader node.

Under the hood of adaptive HA, EdgeDB maintains a state machine to avoid endless switch-overs in an unstable network. State changes only happen when certain conditions are met.

#### Set of possible states:

- **Healthy** - all is good
- **Unhealthy** - a staging state before failover
- **Failover** - backend failover is in process

#### Rules of state switches:

**Unhealthy** -> **Healthy**

- Successfully connected to a non-hot-standby backend.

**Unhealthy** -> **Failover**

- More than 60% (configurable with environment variable `EDGEDB_SERVER_BACKEND_ADAPTIVE_HA_DISCONNECT_PERCENT`) of existing pgcons are “unexpectedly disconnected” (number of existing pgcons is captured at the moment we change to Unhealthy state, and maintained on “expected disconnects” too).
- (and) In Unhealthy state for more than 30 seconds (`EDGEDB_SERVER_BACKEND_ADAPTIVE_HA_UNHEALTHY_MIN_TIME`).
- (and) `sys_pgcon` is down.
- (or) Postgres shutdown/hot-standby notification received.

Healthy -> Unhealthy

- Any unexpected disconnect.

Healthy -> Failover

- Postgres shutdown/hot-standby notification received.

Failover -> Healthy

- Successfully connected to a non-hot-standby backend.
- (and) `sys_pgcon` is healthy.

(“`pgcon`” is a code name for backend connections, and “`sys_pgcon`” is a special backend connection which EdgeDB uses to talk to the “EdgeDB system database”.)

## 8.10 Server configuration

EdgeDB exposes a number of configuration parameters that affect its behavior. In this section we review the ways to change the server configuration, as well as detail each available configuration parameter.

### 8.10.1 Configuring the server

#### 8.10.1.1 EdgeQL

The `configure` command can be used to set the configuration parameters using EdgeQL. For example:

```
edgedb> configure instance set listen_addresses := {'127.0.0.1', '::1'};  
CONFIGURE: OK
```

#### 8.10.1.2 CLI

The `edgedb configure` command allows modifying the system configuration from a terminal:

```
$ edgedb configure set listen_addresses 127.0.0.1 ::1
```

## 8.10.2 Available settings

Below is an overview of available settings. a full reference of settings is available at [Standard Library > Config](#).

### 8.10.2.1 Connection settings

**listen\_addresses** -> **multi str** The TCP/IP address(es) on which the server is to listen for connections from client applications.

**listen\_port** -> **int16** The TCP port the server listens on; defaults to 5656.

### 8.10.2.2 Resource usage

**effective\_ioConcurrency** -> **int64** The number of concurrent disk I/O operations that can be executed simultaneously.

**query\_work\_mem** -> **cfg::memory** The amount of memory used by internal query operations such as sorting.

**shared\_buffers** -> **cfg::memory** The amount of memory used for shared memory buffers.

### 8.10.2.3 Query planning

**default\_statistics\_target** -> **int64** The default data statistics target for the planner.

**effective\_cache\_size** -> **cfg::memory** An estimate of the effective size of the disk cache available to a single query.

### 8.10.2.4 Query behavior

**allow\_bare\_ddl** -> **cfg::AllowBareDDL** Allows for running bare DDL outside a migration. Possible values are `cfg::AllowBareDDL.AlwaysAllow` and `cfg::AllowBareDDL.NeverAllow`.

When you create an instance, this is set to `cfg::AllowBareDDL.AlwaysAllow` until you run a migration. At that point it is set to `cfg::AllowBareDDL.NeverAllow` because it's generally a bad idea to mix migrations with bare DDL.

**apply\_access\_policies** -> **bool** Determines whether access policies should be applied when running queries. Setting this to `false` effectively puts you into super-user mode, ignoring any access policies that might otherwise limit you on the instance.

---

**Note:** This setting can also be conveniently accessed via the “Config” dropdown menu at the top of the EdgeDB UI (accessible by running the CLI command `edgedb ui` from within a project). The setting will apply only to your UI session, so you won’t have to remember to re-enable it when you’re done.

---

### 8.10.2.5 Client connections

**session\_idle\_timeout** → **std::duration** How long client connections can stay inactive before being closed by the server. Defaults to 60 seconds; set to `<duration>'0'` to disable the mechanism.

**session\_idle\_transaction\_timeout** → **std::duration** How long client connections can stay inactive while in a transaction. Defaults to 10 seconds; set to `<duration>'0'` to disable the mechanism.

**query\_execution\_timeout** → **std::duration** How long an individual query can run before being aborted. A value of `<duration>'0'` disables the mechanism; it is disabled by default.

## 8.11 HTTP API

Using HTTP, you may check the health of your EdgeDB instance, check metrics on your instance, and make queries.

Your instance's URL takes the form of `http://<hostname>:<port>/`. For queries, you will append `db/<database-name>/edgeql`.

---

**Note:** Here's how to determine your local EdgeDB instance's HTTP server URL:

- The `hostname` will be `localhost`
- Find the `port` by running `edgedb instance list`. This will print a table of all EdgeDB instances on your machine, including their associated port number.
- In most cases, `database_name` will be `edgedb`. An EdgeDB *instance* can contain multiple databases. On initialization, a default database called `edgedb` is created; all queries are executed against this database unless otherwise specified.

To determine the URL of a remote instance you have linked with the CLI, you can get both the hostname and port of the instance from the “Port” column of the `edgedb instance list` table (formatted as `<hostname>:<port>`). The same guidance on local database names applies here.

---

### 8.11.1 Health Checks

EdgeDB exposes endpoints to check for aliveness and readiness of your database instance.

#### 8.11.1.1 Aliveness

Check if your instance is alive.

```
http://<hostname>:<port>/server/status/alive
```

If your instance is alive, it will respond with a `200` status code and "OK" as the payload. Otherwise, it will respond with a `50x` or a network error.

### 8.11.1.2 Readiness

Check if your instance is ready to receive queries.

```
http://<hostname>:<port>/server/status/ready
```

If your instance is ready, it will respond with a `200` status code and "OK" as the payload. Otherwise, it will respond with a `50x` or a network error.

## 8.11.2 Observability

Retrieve instance metrics.

```
http://<hostname>:<port>/metrics
```

All EdgeDB instances expose a Prometheus-compatible endpoint available via GET request. The following metrics are made available.

### 8.11.2.1 Processes

**`compiler_process_spawns_total` Counter.** Total number of compiler processes spawned.

**`compiler_processes_current` Gauge.** Current number of active compiler processes.

### 8.11.2.2 Backend connections and performance

**`backend_connections_total` Counter.** Total number of backend connections established.

**`backend_connections_current` Gauge.** Current number of active backend connections.

**`backend_connection_establishment_errors_total` Counter.** Number of times the server could not establish a backend connection.

**`backend_connection_establishment_latency` Histogram.** Time it takes to establish a backend connection, in seconds.

**`backend_query_duration` Histogram.** Time it takes to run a query on a backend connection, in seconds.

### 8.11.2.3 Client connections

**`client_connections_total` Counter.** Total number of clients.

**`client_connections_current` Gauge.** Current number of active clients.

**`client_connections_idle_total` Counter.** Total number of forcefully closed idle client connections.

#### 8.11.2.4 Query compilation

`edgeql_query_compilations_total` Counter. Number of compiled/cached queries or scripts.

`edgeql_query_compilation_duration` Histogram. Time it takes to compile an EdgeQL query or script, in seconds.

#### 8.11.2.5 Errors

`background_errors_total` Counter. Number of unhandled errors in background server routines.

### 8.11.3 Querying

Before querying over HTTP, you must first enable the HTTP extension in your schema. Add this to your schema, outside any module:

```
using extension edgeql_http;
```

Then create a new migration and apply it using `edgedb migration create` and `edgedb migrate`, respectively.

Your instance is now able to receive EdgeQL queries over HTTP.

---

**Note:** Enabling the HTTP extension is only required for querying over HTTP. It is *not* required for health checks or observability.

---

#### 8.11.3.1 Making a query request

Make a query to your EdgeDB database using this URL:

```
http://<hostname>:<port>/db/<database-name>/edgeql
```

You may make queries via either the POST or GET HTTP method. Query requests can take the following fields:

- `query` - contains the EdgeQL query string
- `variables` - contains a JSON object where the keys are the parameter names from the query and the values are the arguments to be used in this execution of the query.

When using the GET method, supply `query` and `variables` as query parameters. For a POST request, use the `application/json` content type and submit a JSON payload with `query` and `variables` as top-level keys in that payload as in this example:

Here's an example query you might want to run to insert a new person in your database, as executed from the EdgeDB REPL:

```
db> insert Person { name := <str>$name };
Parameter <str>$name: Pat
{default::Person {id: e9009b00-8d4e-11ed-a556-c7b5bdd6cf7a}}
```

The query inserts a `Person` object. The object's `name` value is parameterized in the query as `$name`.

This GET request would run the same query (assuming the instance is local and the database is named `edgedb`):

```
GET http://localhost:<port>/db/edgedb/edgeql?query=insert%20Person%20%7B%20name%20%3A%3D
%20%3Cstr%3E$name%20%7D%3B&variables=%7B%22name%22%3A%20%22Pat%22%7D
```

As you can see with even this simple query, URL encoding can quickly become onerous with queries over GET.

Here's the JSON payload of a POST request to execute the query:

```
{
  "query": "insert Person { name := <str>$name };",
  "variables": { "name": "Pat" }
}
```

### 8.11.3.2 Response

The response format is the same for both methods. The body of the response is JSON of the following form:

```
{
  "data": [ ... ],
  "error": {
    "message": "Error message",
    "type": "ErrorType",
    "code": 123456
  }
}
```

The `data` response field will contain the response set serialized as a JSON array.

Note that the `error` field will only be present if an error actually occurred. The `error` will further contain the `message` field with the error message string, the `type` field with the name of the type of error and the `code` field with an integer *error code*.

---

**Note:** Caution is advised when reading `decimal` or `bigint` values using the HTTP protocol because the results are provided in JSON format. The JSON specification does not have a limit on significant digits, so a `decimal` or a `bigint` number can be losslessly represented in JSON. However, JSON decoders in many languages will read all such numbers as some kind of 32- or 64-bit number type, which may result in errors or precision loss. If such loss is unacceptable, then consider casting the value into `str` and decoding it on the client side into a more appropriate type.

---

## 8.12 SQL support

### 8.12.1 Connecting

EdgeDB supports running read-only SQL queries via the Postgres protocol to enable connecting EdgeDB to existing BI and analytics solutions. Any Postgres-compatible client can connect to your EdgeDB database by using the same port that is used for the EdgeDB protocol and the same database name, username, and password you already use for your database.

Here's how you might connect to a local instance on port 10701 (determined by running `edgedb instance list`) with a database `edgedb` using the `psql` CLI:

```
$ psql -h localhost -p 10701 -U edgedb -d edgedb
```

You'll then be prompted for a password. If you don't have it, you can run `edgedb instance credentials --insecure-dsn` and grab it out of the DSN the command returns. (It's the string between the second colon and the “at” symbol: `edgedb://edgedb:PASSWORD_IS_HERE@<host>:<port>/<database>`)

This works well to test SQL support, but if you are going to be using it on an ongoing basis, you may want to create a new role and use it to authenticate your SQL clients. Set a password when you create your role. Then, use the role name as your user name when you connect via your SQL client.

```
create superuser role sql {
    set password := 'your-password'
};
```

```
$ psql -h localhost -p 10701 -U sql -d edgedb
```

In this example, when prompted for the password, you would enter `your-password`.

**Warning:** EdgeDB server requires TLS by default, and this is also true for our SQL support. Make sure to require SSL encryption in your SQL tool or client when using EdgeDB’s SQL support. Alternatively, you can disable the TLS requirement by setting the `EDGEDB_SERVER_BINARY_ENDPOINT_SECURITY` environment variable to optional.

## 8.12.2 Querying

Object types in your EdgeDB schema are exposed as regular SQL tables containing all the data you store in your EdgeDB database.

If you have a database with the following schema:

```
module default {
    type Person {
        name: str;
    };

    type Movie extending common::Content {
        release_year: int32;
        director: Person;
        star: Person {
            role: str;
        };
        multi actors: Person {
            role: str;
        };
        multi labels: str;
    };
}

module common {
    type Content {
        title: str;
    };
}
```

you can access your data after connecting using the following SQL queries:

```
SELECT id, name FROM "Person";
SELECT id, title, release_year, director_id, star_id FROM "Movie";
```

Because the link `star` has link properties, it has its own table. `source` is the `id` of the `Movie`. `target` is the `id` of the `Person`.

```
SELECT source, target, role FROM "Movie.star";
```

Links are in separate tables.

```
SELECT source, target, role FROM "Movie.actors";
```

Multi properties are in separate tables. `source` is the `id` of the `Movie`. `target` is the value of the property.

```
SELECT source, target FROM "Movie.labels";
```

When types are extended, parent object types' tables will by default contain all objects of both the type and any types extended by it. The query below will return all `common::Content` objects as well as all `Movie` objects.

```
SELECT id, title FROM common."Content";
```

To omit objects of extended types, use `ONLY`. This query will return `common::Content` objects but not `Movie` objects.

```
SELECT id, title FROM ONLY common."Content";
```

The SQL connector supports read-only statements and will throw errors if the client attempts `INSERT`, `UPDATE`, `DELETE`, or any `DDL` command. It supports all SQL expressions supported by Postgres.

```
SELECT id, 'Title is: ' || tittle
FROM "Movie" m
JOIN "Person" d ON m.director_id = d.id
WHERE EXISTS (
    SELECT 1
    FROM "Movie.actors" act
    WHERE act.source = m.id
);
```

EdgeDB accomplishes this by emulating the `information_schema` and `pg_catalog` views to mimic the catalogs provided by Postgres 13.

---

**Note:** Learn more about the Postgres information schema from [the Postgres information schema documentation](#).

---

**Warning:** Some tables may be truncated and may not contain all objects you would expect a true Postgres instance to contain. This may be a source of problems when using tools that introspect the database and rely on internal Postgres features.

### 8.12.3 Tested SQL tools

- [pg\\_dump](#)
- [Metabase](#)
- [Cluvio](#)
- [Tableau](#)
- [DataGrip](#)
- [Airbyte<sup>1</sup>](#)
- [Fivetran<sup>1</sup>](#)
- [Hevo<sup>1</sup>](#)
- [Stitch<sup>1</sup>](#)
- [dbt<sup>2</sup>](#)

## 8.13 Binary protocol

EdgeDB uses a message-based binary protocol for communication between clients and servers. The protocol is supported over TCP/IP.

---

<sup>1</sup> At the moment, EdgeDB does not support “Log replication” (i.e., using the [Postgres replication mechanism](#)). Supported replication methods include [XMIN Replication](#), incremental updates using “a user-defined monotonically increasing id,” and full table updates.

<sup>2</sup> dbt models are built and stored in the database as either tables or views. Because the EdgeDB SQL connector does not allow writing or even creating schemas, view, or tables, any attempt to materialize dbt models will result in errors. If you want to build the models, we suggest first transferring your data to a true Postgres instance via pg\_dump or Airbyte. Tests and previews can still be run directly against the EdgeDB instance.

### 8.13.1 Messages

Server Messages	
<i>AuthenticationOK</i>	Authentication is successful.
<i>AuthenticationSASL</i>	SASL authentication is required.
<i>AuthenticationSASLContinue</i>	SASL authentication challenge.
<i>AuthenticationSASLFinal</i>	SASL authentication final message.
<i>CommandComplete</i>	Successful completion of a command.
<i>CommandDataDescription</i>	Description of command data input and output.
<i>StateDataDescription</i>	Description of state data.
<i>Data</i>	Command result data element.
<i>Dump Header</i>	Initial message of the database backup protocol
<i>Dump Block</i>	Single chunk of database backup data
<i>ErrorResponse</i>	Server error.
<i>LogMessage</i>	Server log message.
<i>ParameterStatus</i>	Server parameter value.
<i>ReadyForCommand</i>	Server is ready for a command.
<i>RestoreReady</i>	Successful response to the <i>Restore</i> message
<i>ServerHandshake</i>	Initial server connection handshake.
<i>ServerKeyData</i>	Opaque token identifying the server connection.
Client Messages	
<i>AuthenticationSASLInitialResponse</i>	SASL authentication initial response.
<i>AuthenticationSASLResponse</i>	SASL authentication response.
<i>ClientHandshake</i>	Initial client connection handshake.
<i>Dump</i>	Initiate database backup
<i>Parse</i>	Parse EdgeQL command(s).
<i>Execute</i>	Parse and/or execute a query.
<i>Restore</i>	Initiate database restore
<i>RestoreBlock</i>	Next block of database dump
<i>RestoreEof</i>	End of database dump
<i>Sync</i>	Provide an explicit synchronization point.
<i>Terminate</i>	Terminate the connection.

#### 8.13.1.1 ErrorResponse

Sent by: server.

Format:

```
struct ErrorResponse {
    // Message type ('E').
    uint8      mtype = 0x45;

    // Length of message contents in bytes,
    // including self.
    uint32     message_length;

    // Message severity.
    uint8<ErrorSeverity> severity;

    // Message code.
}
```

(continues on next page)

(continued from previous page)

```

    uint32          error_code;

    // Error message.
    string          message;

    // Error attributes.
    uint16          num_attributes;
    KeyValue        attributes[num_attributes];
};

```

```

enum ErrorSeverity {
    ERROR          = 0x78;
    FATAL          = 0xc8;
    PANIC          = 0xff;
};

```

See the [list of error codes](#) for all possible error codes.

Known headers:

- 0x0001 HINT: str – error hint.
- 0x0002 DETAILS: str – error details.
- 0x0101 SERVER\_TRACEBACK: str – error traceback from server (is only sent in dev mode).
- 0xFFFF1 POSITION\_START – byte offset of the start of the error span.
- 0xFFFF2 POSITION\_END – byte offset of the end of the error span.
- 0xFFFF3 LINE\_START – one-based line number of the start of the error span.
- 0xFFFF4 COLUMN\_START – one-based column number of the start of the error span.
- 0xFFFF5 UTF16\_COLUMN\_START – zero-based column number in UTF-16 encoding of the start of the error span.
- 0xFFFF6 LINE\_END – one-based line number of the start of the error span.
- 0xFFFF7 COLUMN\_END – one-based column number of the start of the error span.
- 0xFFFF8 UTF16\_COLUMN\_END – zero-based column number in UTF-16 encoding of the end of the error span.
- 0xFFFF9 CHARACTER\_START – zero-based offset of the error span in terms of Unicode code points.
- 0xFFFFA CHARACTER\_END – zero-based offset of the end of the error span.

Notes:

1. Error span is the range of characters (or equivalent bytes) of the original query that compiler points to as the source of the error.
2. COLUMN\_\* is defined in terms of width of characters defined by Unicode Standard Annex #11, in other words, the column number in the text if rendered with monospace font, e.g. in a terminal.
3. UTF16\_COLUMN\_\* is defined as number of UTF-16 code units (i.e. two byte-pairs) that precede target character on the same line.
4. \*\_END points to a next character after the last character of the error span.

### 8.13.1.2 LogMessage

Sent by: server.

Format:

```
struct LogMessage {
    // Message type ('L').
    uint8          mtype = 0x4c;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Message severity.
    uint8<MessageSeverity> severity;

    // Message code.
    uint32         code;

    // Message text.
    string         text;

    // Message annotations.
    uint16         num_annotations;
    Annotation     annotations[num_annotations];
};
```

```
enum MessageSeverity {
    DEBUG        = 0x14;
    INFO         = 0x28;
    NOTICE        = 0x3c;
    WARNING       = 0x50;
};
```

See the [list of error codes](#) for all possible log message codes.

### 8.13.1.3 ReadyForCommand

Sent by: server.

Format:

```
struct ReadyForCommand {
    // Message type ('Z').
    uint8          mtype = 0x5a;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // A set of annotations.
    uint16         num_annotations;
    Annotation     annotations[num_annotations];
```

(continues on next page)

(continued from previous page)

```
// Transaction state.
uint8<TransactionState> transaction_state;
};
```

```
enum TransactionState {
    NOT_IN_TRANSACTION = 0x49;
    IN_TRANSACTION = 0x54;
    IN_FAILED_TRANSACTION = 0x45;
};
```

#### 8.13.1.4 RestoreReady

Sent by: server.

Initial *Restore* message accepted, ready to receive data. See *Restore Database Flow*.

Format:

```
struct RestoreReady {
    // Message type ('+').
    uint8          mtype = 0x2b;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // A set of annotations.
    uint16         num_annotations;
    Annotation     annotations[num_annotations];

    // Number of parallel jobs for restore,
    // currently always "1"
    uint16         jobs;
};
```

#### 8.13.1.5 CommandComplete

Sent by: server.

Format:

```
struct CommandComplete {
    // Message type ('C').
    uint8          mtype = 0x43;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // A set of annotations.
```

(continues on next page)

(continued from previous page)

```

  uint16      num_annotations;
Annotation    annotations[num_annotations];

// A bit mask of allowed capabilities.
  uint64<Capability> capabilities;

// Command status.
  string      status;

// State data descriptor ID.
  uuid        state_typedesc_id;

// Encoded state data.
  bytes       state_data;
};

```

### 8.13.1.6 Dump

Sent by: client.

Initiates a database backup. See [Dump Database Flow](#).

Format:

```

struct Dump {
  // Message type ('>').
  uint8      mtype = 0x3e;

  // Length of message contents in bytes,
// including self.
  uint32     message_length;

// A set of annotations.
  uint16      num_annotations;
Annotation    annotations[num_annotations];
};

```

### 8.13.1.7 CommandDataDescription

Sent by: server.

Format:

```

struct CommandDataDescription {
  // Message type ('T').
  uint8      mtype = 0x54;

  // Length of message contents in bytes,
// including self.
  uint32     message_length;

```

(continues on next page)

(continued from previous page)

```
// A set of annotations.
uint16           num_annotations;
Annotation       annotations[num_annotations];

// A bit mask of allowed capabilities.
uint64<Capability> capabilities;

// Actual result cardinality.
uint8<Cardinality> result_cardinality;

// Argument data descriptor ID.
uuid             input_typedesc_id;

// Argument data descriptor.
bytes            input_typedesc;

// Output data descriptor ID.
uuid             output_typedesc_id;

// Output data descriptor.
bytes            output_typedesc;
};

};
```

```
enum Cardinality {
    NO_RESULT          = 0x6e;
    AT_MOST_ONE        = 0x6f;
    ONE                = 0x41;
    MANY               = 0x6d;
    AT_LEAST_ONE       = 0x4d;
};
```

The format of the `input_typedesc` and `output_typedesc` fields is described in the [Type descriptors](#) section.

### 8.13.1.8 StateDataDescription

Sent by: server.

## Format:

```
struct StateDataDescription {
    // Message type ('s').
    uint8          mtype = 0x7f

    // Length of message content
    // including self.
    uint32         message_len

    // Updated state data descriptor
    uuid           typedesc_id

    // State data descriptor.
}
```

---

(continues on next page)

(continued from previous page)

```
bytes          typedesc;
};
```

The format of the *typedesc* fields is described in the [Type descriptors](#) section.

### 8.13.1.9 Sync

Sent by: client.

Format:

```
struct Sync {
    // Message type ('S').
    uint8      mtype = 0x53;

    // Length of message contents in bytes,
    // including self.
    uint32     message_length;
};
```

### 8.13.1.10 Restore

Sent by: client.

Initiate restore to the current database. See [Restore Database Flow](#).

Format:

```
struct Restore {
    // Message type ('<').
    uint8      mtype = 0x3c;

    // Length of message contents in bytes,
    // including self.
    uint32     message_length;

    // A set of key-value pairs.
    uint16     num_attributes;
    KeyValue   attributes[num_attributes];

    // Number of parallel jobs for restore
    // (only "1" is supported)
    uint16     jobs;

    // Original DumpHeader packet data
    // excluding mtype and message_length
    bytes      header_data;
};
```

### 8.13.1.11 RestoreBlock

Sent by: client.

Send dump file data block. See [Restore Database Flow](#).

Format:

```
struct RestoreBlock {
    // Message type ('=').
    uint8          mtype = 0x3d;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Original DumpBlock packet data excluding
    // mtype and message_length
    bytes          block_data;
};
```

### 8.13.1.12 RestoreEof

Sent by: client.

Notify server that dump is fully uploaded. See [Restore Database Flow](#).

Format:

```
struct RestoreEof {
    // Message type ('.') .
    uint8          mtype = 0x2e;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;
};
```

### 8.13.1.13 Execute

Sent by: client.

Format:

```
struct Execute {
    // Message type ('0') .
    uint8          mtype = 0x4f;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // A set of annotations.
    uint16         num_annotations;
```

(continues on next page)

(continued from previous page)

```

Annotation      annotations[num_annotations];

// A bit mask of allowed capabilities.
uint64<Capability> allowed_capabilities;

// A bit mask of query options.
uint64<CompilationFlag> compilation_flags;

// Implicit LIMIT clause on returned sets.
uint64          implicit_limit;

// Data output format.
uint8<OutputFormat> output_format;

// Expected result cardinality.
uint8<Cardinality> expected_cardinality;

// Command text.
string          command_text;

// State data descriptor ID.
uuid            state_typedesc_id;

// Encoded state data.
bytes           state_data;

// Argument data descriptor ID.
uuid            input_typedesc_id;

// Output data descriptor ID.
uuid            output_typedesc_id;

// Encoded argument data.
bytes           arguments;
};
```

```

enum OutputFormat {
    BINARY        = 0x62;
    JSON          = 0x6a;
    JSON_ELEMENTS = 0x4a;
    NONE          = 0x6e;
};
```

Use:

- BINARY to return data encoded in binary.
- JSON to return data as single row and single field that contains the resultset as a single JSON array”.
- JSON\_ELEMENTS to return a single JSON string per top-level set element. This can be used to iterate over a large result set efficiently.
- NONE to prevent the server from returning data, even if the EdgeQL command does.

The data in *arguments* must be encoded as a *tuple value* described by a type descriptor identified by *input\_typedesc\_id*.

```
enum Cardinality {
    NO_RESULT      = 0x6e;
    AT_MOST_ONE    = 0x6f;
    ONE            = 0x41;
    MANY           = 0x6d;
    AT_LEAST_ONE   = 0x4d;
};
```

### 8.13.1.14 Parse

Sent by: client.

```
struct Parse {
    // Message type ('P').
    uint8          mtype = 0x50;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // A set of annotations.
    uint16         num_annotations;
    Annotation     annotations[num_annotations];

    // A bit mask of allowed capabilities.
    uint64<Capability> allowed_capabilities;

    // A bit mask of query options.
    uint64<CompilationFlag> compilation_flags;

    // Implicit LIMIT clause on returned sets.
    uint64         implicit_limit;

    // Data output format.
    uint8<OutputFormat> output_format;

    // Expected result cardinality.
    uint8<Cardinality> expected_cardinality;

    // Command text.
    string         command_text;

    // State data descriptor ID.
    uuid           state_typedesc_id;

    // Encoded state data.
    bytes          state_data;
};
```

```
enum Capability {
    MODIFICATIONS = 0x1;
```

(continues on next page)

(continued from previous page)

```
SESSION_CONFIG = 0x2;
TRANSACTION = 0x4;
DDL = 0x8;
PERSISTENT_CONFIG = 0x10;
ALL = 0xffffffffffffffffffff;
};
```

See [RFC1004](#) for more information on capability flags.

```
enum CompilationFlag {
    INJECT_OUTPUT_TYPE_IDS = 0x1;
    INJECT_OUTPUT_TYPE_NAMES = 0x2;
    INJECT_OUTPUT_OBJECT_IDS = 0x4;
};
```

Use:

- 0x0000\_0000\_0000\_0001 (INJECT\_OUTPUT\_TYPE\_IDS) – if set, all returned objects have a `__tid__` property set to their type ID (equivalent to having an implicit `__tid__ := __type__.id` computed property.)
- 0x0000\_0000\_0000\_0002 (INJECT\_OUTPUT\_TYPE\_NAMES) – if set all returned objects have a `__tname__` property set to their type name (equivalent to having an implicit `__tname__ := __type__.name` computed property.) Note that specifying this flag might slow down queries.
- 0x0000\_0000\_0000\_0004 (INJECT\_OUTPUT\_OBJECT\_IDS) – if set all returned objects have an `id` property set to their identifier, even if not specified explicitly in the output shape.

```
enum OutputFormat {
    BINARY = 0x62;
    JSON = 0x6a;
    JSON_ELEMENTS = 0x4a;
    NONE = 0x6e;
};
```

Use:

- BINARY to return data encoded in binary.
- JSON to return data as single row and single field that contains the resultset as a single JSON array”.
- JSON\_ELEMENTS to return a single JSON string per top-level set element. This can be used to iterate over a large result set efficiently.
- NONE to prevent the server from returning data, even if the EdgeQL statement does.

```
enum Cardinality {
    NO_RESULT = 0x6e;
    AT_MOST_ONE = 0x6f;
    ONE = 0x41;
    MANY = 0x6d;
    AT_LEAST_ONE = 0x4d;
};
```

### 8.13.1.15 Data

Sent by: server.

Format:

```
struct Data {
    // Message type ('D').
    uint8          mtype = 0x44;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Encoded output data array. The array is
    // currently always of size 1.
    uint16         num_data;
    DataElement    data[num_data];
};
```

```
struct DataElement {
    // Encoded output data.
    uint32         num_data;
    uint8          data[num_data];
};
```

The exact encoding of `DataElement.data` is defined by the query output *type descriptor*.

Wire formats for the standard scalar types and collections are documented in *Data wire formats*.

### 8.13.1.16 Dump Header

Sent by: server.

Initial message of database backup protocol. See *Dump Database Flow*.

Format:

```
struct DumpHeader {
    // Message type ('@').
    uint8          mtype = 0x40;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // A set of key-value pairs.
    uint16         num_attributes;
    KeyValue      attributes[num_attributes];

    // Major version of EdgeDB.
    uint16         major_ver;

    // Minor version of EdgeDB.
```

(continues on next page)

(continued from previous page)

```

    uint16          minor_ver;

    // Schema.
    string          schema_ddl;

    // Type identifiers.
    uint32          num_types;
    DumpTypeInfo    types[num_types];

    // Object descriptors.
    uint32          num_descriptors;
    DumpObjectDesc  descriptors[num_descriptors];
};

```

```

struct DumpTypeInfo {
    string          type_name;

    string          type_class;

    uuid            type_id;
};

```

```

struct DumpObjectDesc {
    uuid            object_id;

    bytes           description;

    uint16          num_dependencies;
    uuid            dependencies[num_dependencies];
};

```

Known headers:

- 101 BLOCK\_TYPE – block type, always “T”
- 102 SERVER\_TIME – server time when dump is started as a floating point unix timestamp stringified
- 103 SERVER\_VERSION – full version of server as string
- 105 SERVER\_CATALOG\_VERSION – the catalog version of the server, as a 64-bit integer. The catalog version is an identifier that is incremented whenever a change is made to the database layout or standard library.

### 8.13.1.17 Dump Block

Sent by: server.

The actual protocol data in the backup protocol. See [Dump Database Flow](#).

Format:

```

struct DumpBlock {
    // Message type ('=').
    uint8            mtype = 0x3d;
}

```

(continues on next page)

(continued from previous page)

```
// Length of message contents in bytes,
// including self.
uint32          message_length;

// A set of key-value pairs.
uint16          num_attributes;
KeyValue        attributes[num_attributes];
};
```

Known headers:

- 101 BLOCK\_TYPE – block type, always “D”
- 110 BLOCK\_ID – block identifier (16 bytes of UUID)
- 111 BLOCK\_NUM – integer block index stringified
- 112 BLOCK\_DATA – the actual block data

### 8.13.1.18 ServerKeyData

Sent by: server.

Format:

```
struct ServerKeyData {
    // Message type ('K').
    uint8          mtype = 0x4b;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Key data.
    uint8          data[32];
};
```

### 8.13.1.19 ParameterStatus

Sent by: server.

Format:

```
struct ParameterStatus {
    // Message type ('S').
    uint8          mtype = 0x53;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Parameter name.
    bytes          name;
```

(continues on next page)

(continued from previous page)

```
// Parameter value.
bytes          value;
};
```

Known statuses:

- `suggested_pool_concurrency` – suggested default size for clients connection pools. Serialized as UTF-8 encoded string.
- `system_config` – a set of instance-level configuration settings exposed to clients on connection. Serialized as:

```
struct ParameterStatus_SystemConfig {
    // Type descriptor prefixed with type
    // descriptor uuid.
    uint32      num_typedesc;
    uint8       typedesc[num_typedesc];

    // Configuration settings data.
    DataElement  data[1];
};
```

Where `DataElement` is defined in the same way as for the `Data` message:

```
struct DataElement {
    // Encoded output data.
    uint32      num_data;
    uint8       data[num_data];
};
```

### 8.13.1.20 ClientHandshake

Sent by: client.

Format:

```
struct ClientHandshake {
    // Message type ('V').
    uint8      mtype = 0x56;

    // Length of message contents in bytes,
    // including self.
    uint32      message_length;

    // Requested protocol major version.
    uint16      major_ver;

    // Requested protocol minor version.
    uint16      minor_ver;

    // Connection parameters.
    uint16      num_params;
    ConnectionParam params[num_params];
```

(continues on next page)

(continued from previous page)

```
// Requested protocol extensions.
uint16          num_extensions;
ProtocolExtension extensions[num_extensions];
};
```

```
struct ConnectionParam {
    string        name;
    string        value;
};
```

```
struct ProtocolExtension {
    // Extension name.
    string        name;

    // A set of extension annotations.
    uint16        num_annotations;
    Annotation   annotations[num_annotations];
};
```

The `ClientHandshake` message is the first message sent by the client upon connecting to the server. It is the first phase of protocol negotiation, where the client sends the requested protocol version and extensions. Currently, the only defined `major_ver` is 1, and `minor_ver` is 0. No protocol extensions are currently defined. The server always responds with the `ServerHandshake`.

### 8.13.1.21 ServerHandshake

Sent by: server.

Format:

```
struct ServerHandshake {
    // Message type ('v').
    uint8          mtype = 0x76;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // maximum supported or client-requested
    // protocol major version, whichever is
    // greater.
    uint16         major_ver;

    // maximum supported or client-requested
    // protocol minor version, whichever is
    // greater.
    uint16         minor_ver;

    // Supported protocol extensions.
```

(continues on next page)

(continued from previous page)

```
    uint16      num_extensions;
    ProtocolExtension extensions[num_extensions];
};
```

```
struct ProtocolExtension {
    // Extension name.
    string      name;

    // A set of extension annotations.
    uint16      num_annotations;
    Annotation  annotations[num_annotations];
};
```

The `ServerHandshake` message is a direct response to the `ClientHandshake` message and is sent by the server in the case where the server does not support the protocol version or protocol extensions requested by the client. It contains the maximum protocol version supported by the server, considering the version requested by the client. It also contains the intersection of the client-requested and server-supported protocol extensions. Any requested extensions not listed in the `Server Handshake` message are considered unsupported.

### 8.13.1.22 AuthenticationOK

Sent by: server.

Format:

```
struct AuthenticationOK {
    // Message type ('R').
    uint8      mtype = 0x52;

    // Length of message contents in bytes,
    // including self.
    uint32      message_length;

    // Specifies that this message contains a
    // successful authentication indicator.
    uint32      auth_status;
};
```

The `AuthenticationOK` message is sent by the server once it considers the authentication to be successful.

### 8.13.1.23 AuthenticationSASL

Sent by: server.

Format:

```
struct AuthenticationRequiredSASLMessage {
    // Message type ('R').
    uint8      mtype = 0x52;

    // Length of message contents in bytes,
    // including self.
```

(continues on next page)

(continued from previous page)

```

uint32      message_length;

// Specifies that this message contains a
// SASL authentication request.
uint32      auth_status = 0xa;

// A list of supported SASL authentication
// methods.
uint32      num_methods;
string      methods[num_methods];
};

```

The `AuthenticationSASL` message is sent by the server if it determines that a SASL-based authentication method is required in order to connect using the connection parameters specified in the `ClientHandshake`. The message contains a list of *authentication methods* supported by the server in the order preferred by the server.

**Note:** At the moment, the only SASL authentication method supported by EdgeDB is SCRAM-SHA-256 ([RFC 7677](#)).

The client must select an appropriate authentication method from the list returned by the server and send an `AuthenticationSASLInitialResponse`. One or more server-challenge and client-response message follow. Each server-challenge is sent in an `AuthenticationSASLContinue`, followed by a response from the client in an `AuthenticationSASLResponse` message. The particulars of the messages are mechanism specific. Finally, when the authentication exchange is completed successfully, the server sends an `AuthenticationSASLFinal`, followed immediately by an `AuthenticationOK`.

#### 8.13.1.24 AuthenticationSASLContinue

Sent by: server.

Format:

```

struct AuthenticationSASLContinue {
    // Message type ('R').
    uint8      mtype = 0x52;

    // Length of message contents in bytes,
// including self.
    uint32      message_length;

    // Specifies that this message contains a
// SASL challenge.
    uint32      auth_status = 0xb;

    // Mechanism-specific SASL data.
    bytes      sasl_data;
};

```

### 8.13.1.25 AuthenticationSASLFinal

Sent by: server.

Format:

```
struct AuthenticationSASLFinal {
    // Message type ('R').
    uint8          mtype = 0x52;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Specifies that SASL authentication has
    // completed.
    uint32         auth_status = 0xc;

    bytes          sasl_data;
};
```

### 8.13.1.26 AuthenticationSASLInitialResponse

Sent by: client.

Format:

```
struct AuthenticationSASLInitialResponse {
    // Message type ('p').
    uint8          mtype = 0x70;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Name of the SASL authentication
    // mechanism that the client selected.
    string         method;

    // Mechanism-specific "Initial Response"
    // data.
    bytes          sasl_data;
};
```

### 8.13.1.27 AuthenticationSASLResponse

Sent by: client.

Format:

```
struct AuthenticationSASLResponse {
    // Message type ('r').
    uint8          mtype = 0x72;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;

    // Mechanism-specific response data.
    bytes          sasl_data;
};
```

### 8.13.1.28 Terminate

Sent by: client.

Format:

```
struct Terminate {
    // Message type ('X').
    uint8          mtype = 0x58;

    // Length of message contents in bytes,
    // including self.
    uint32         message_length;
};
```

## 8.13.2 Errors

### 8.13.2.1 Errors inheritance

Each error in EdgeDB consists of a code, a name, and optionally tags. Errors in EdgeDB can inherit from other errors. This is denoted by matching code prefixes. For example, `TransactionConflictError (0x_05_03_01_00)` is the parent error for `TransactionSerializationError (0x_05_03_01_01)` and `TransactionDeadlockError (0x_05_03_01_02)`. The matching prefix here is `0x_05_03_01`.

When the EdgeDB client expects a more general error and EdgeDB returns a more specific error that inherits from the general error, the check in the client must take this into account. This can be expressed by the `binary` and `operation` or `&` operator in most programming languages:

```
(expected_error_code & server_error_code) == expected_error_code
```

Note that although it is not explicitly stated in the `edb/api/errors.txt` file, each inherited error must contain all tags of the parent error. Given that, `TransactionSerializationError` and `TransactionDeadlockError`, for example, must contain the `SHOULD_RETRY` tag that is defined for `TransactionConflictError`.

### 8.13.2.2 Error codes

Error codes and names as specified in `edb/api/errors.txt`:

### 8.13.3 Type descriptors

This section describes how type information for query input and results is encoded. Specifically, this is needed to decode the server response to the `CommandDataDescription` message.

The type descriptor is essentially a list of type information *blocks*:

- each *block* encodes one type;
- *blocks* can reference other *blocks*.

While parsing the *blocks*, a database driver can assemble an *encoder* or a *decoder* of the EdgeDB binary data.

An *encoder* is used to encode objects, native to the driver's runtime, to binary data that EdgeDB can decode and work with.

A *decoder* is used to decode data from EdgeDB native format to data types native to the driver.

There is one special type with *type id* of zero: `00000000-0000-0000-0000-000000000000`. The describe result of this type contains zero *blocks*. It's used when a statement returns no meaningful results, e.g. the `CREATE DATABASE` example statement. It is also used to represent the input descriptor when a query does not receive any arguments, or the state descriptor for an empty/default state.

#### 8.13.3.1 Set Descriptor

```
struct SetDescriptor {
    // Indicates that this is a Set value descriptor.
    uint8    type = 0;

    // Descriptor ID.
    uuid      id;

    // Set element type descriptor index.
    uint16   type_pos;
};
```

Set values are encoded on the wire as *single-dimensional arrays*.

#### 8.13.3.2 Object Shape Descriptor

```
struct ObjectShapeDescriptor {
    // Indicates that this is an
    // Object Shape descriptor.
    uint8        type = 1;

    // Descriptor ID.
    uuid         id;

    // Number of elements in shape.
    uint16       element_count;
```

(continues on next page)

(continued from previous page)

```

    ShapeElement elements[element_count];
};

struct ShapeElement {
    // Field flags:
    // 1 << 0: the field is implicit
    // 1 << 1: the field is a link property
    // 1 << 2: the field is a link
    uint32 flags;

    uint8<Cardinality> cardinality;

    // Field name.
    string name;

    // Field type descriptor index.
    uint16 type_pos;
};

```

```

enum Cardinality {
    NO_RESULT = 0x6e;
    AT_MOST_ONE = 0x6f;
    ONE = 0x41;
    MANY = 0x6d;
    AT_LEAST_ONE = 0x4d;
};

```

Objects are encoded on the wire as *tuples*.

#### 8.13.3.3 Base Scalar Type Descriptor

```

struct BaseScalarTypeDescriptor {
    // Indicates that this is an
    // Base Scalar Type descriptor.
    uint8 type = 2;

    // Descriptor ID.
    uuid id;
};

```

The descriptor IDs for base scalar types are constant. The following table lists all EdgeDB base types descriptor IDs:

ID	Type
00000000-0000-0000-0000-0000000000100	<i>std::uuid</i>
00000000-0000-0000-0000-0000000000101	<i>std::str</i>
00000000-0000-0000-0000-0000000000102	<i>std::bytes</i>
00000000-0000-0000-0000-0000000000103	<i>std::int16</i>
00000000-0000-0000-0000-0000000000104	<i>std::int32</i>
00000000-0000-0000-0000-0000000000105	<i>std::int64</i>
00000000-0000-0000-0000-0000000000106	<i>std::float32</i>
00000000-0000-0000-0000-0000000000107	<i>std::float64</i>
00000000-0000-0000-0000-0000000000108	<i>std::decimal</i>
00000000-0000-0000-0000-0000000000109	<i>std::bool</i>
00000000-0000-0000-0000-000000000010A	<i>std::datetime</i>
00000000-0000-0000-0000-000000000010E	<i>std::duration</i>
00000000-0000-0000-0000-000000000010F	<i>std::json</i>
00000000-0000-0000-0000-000000000010B	<i>cal::local_datetime</i>
00000000-0000-0000-0000-000000000010C	<i>cal::local_date</i>
00000000-0000-0000-0000-000000000010D	<i>cal::local_time</i>
00000000-0000-0000-0000-0000000000110	<i>std::bigint</i>
00000000-0000-0000-0000-0000000000111	<i>cal::relative_duration</i>
00000000-0000-0000-0000-0000000000112	<i>cal::date_duration</i>
00000000-0000-0000-0000-0000000000130	<i>cfg::memory</i>

#### 8.13.3.4 Scalar Type Descriptor

```
struct ScalarTypeDescriptor {
    // Indicates that this is a
    // Scalar Type descriptor.
    uint8 type = 3;

    // Descriptor ID.
    uuid id;

    // Parent type descriptor index.
    uint16 base_type_pos;
};
```

#### 8.13.3.5 Tuple Type Descriptor

```
struct TupleTypeDescriptor {
    // Indicates that this is a
    // Tuple Type descriptor.
    uint8 type = 4;

    // Descriptor ID.
    uuid id;

    // The number of elements in tuple.
    uint16 element_count;
```

(continues on next page)

(continued from previous page)

```
// Indexes of element type descriptors.
uint16    element_types[element_count];
};
```

An empty tuple type descriptor has an ID of 00000000-0000-0000-0000-0000000000FF.

#### 8.13.3.6 Named Tuple Type Descriptor

```
struct NamedTupleTypeDescriptor {
    // Indicates that this is a
    // Named Tuple Type descriptor.
    uint8      type = 5;

    // Descriptor ID.
    uuid       id;

    // The number of elements in tuple.
    uint16     element_count;

    // Indexes of element type descriptors.
    TupleElement elements[element_count];
};

struct TupleElement {
    // Field name.
    string     name;

    // Field type descriptor index.
    int16     type_pos;
};
```

#### 8.13.3.7 Array Type Descriptor

```
struct ArrayTypeDescriptor {
    // Indicates that this is an
    // Array Type descriptor.
    uint8      type = 6;

    // Descriptor ID.
    uuid       id;

    // Element type descriptor index.
    uint16     type_pos;

    // The number of array dimensions, at least 1.
    uint16     dimension_count;

    // Sizes of array dimensions, -1 indicates
    // unbound dimension.
};
```

(continues on next page)

(continued from previous page)

```
    uint32      dimensions[dimension_count];
};
```

### 8.13.3.8 Enumeration Type Descriptor

```
struct EnumerationTypeDescriptor {
    // Indicates that this is an
    // Enumeration Type descriptor.
    uint8      type = 7;

    // Descriptor ID.
    uuid      id;

    // The number of enumeration members.
    uint16     member_count;

    // Names of enumeration members.
    string     members[member_count];
};
```

### 8.13.3.9 Input Shape Descriptor

```
struct InputShapeDescriptor {
    // Indicates that this is an
    // Object Shape descriptor.
    uint8      type = 8;

    // Descriptor ID.
    uuid      id;

    // Number of elements in shape.
    uint16     element_count;

    ShapeElement   elements[element_count];
};
```

Input objects are encoded on the wire as *sparse objects*.

### 8.13.3.10 Range Type Descriptor

```
struct RangeTypeDescriptor {
    // Indicates that this is a
    // Range Type descriptor.
    uint8      type = 9;

    // Descriptor ID.
    uuid      id;
```

(continues on next page)

(continued from previous page)

```
// Range type descriptor index.  
uint16      type_pos;  
};
```

Ranges are encoded on the wire as *ranges*.

#### 8.13.3.11 Scalar Type Name Annotation

Part of the type descriptor when the *Execute* client message has the `INLINE_TYPENAMES` header set. Every non-built-in base scalar type and all enum types would have their full schema name provided via this annotation.

```
struct TypeAnnotationDescriptor {  
    uint8      type = 0xff;  
  
    // ID of the scalar type.  
    uuid      id;  
  
    // Type name.  
    string      type_name;  
};
```

#### 8.13.3.12 Type Annotation Descriptor

Drivers must ignore unknown type annotations.

```
struct TypeAnnotationDescriptor {  
    // Indicates that this is an  
    // Type Annotation descriptor.  
    uint8      type = 0x80..0xfe;  
  
    // ID of the descriptor the  
    // annotation is for.  
    uuid      id;  
  
    // Annotation text.  
    string      annotation;  
};
```

### 8.13.4 Data wire formats

This section describes the data wire format of standard EdgeDB types.

### 8.13.4.1 Sets and array<>

The set and array values are represented as the following structure:

```
struct SetOrArrayValue {
    // Number of dimensions, currently must
    // always be 0 or 1. 0 indicates an empty set or array.
    int32      ndims;

    // Reserved.
    int32      reserved0;

    // Reserved.
    int32      reserved1;

    // Dimension data.
    Dimension  dimensions[ndims];

    // Element data, the number of elements
    // in this array is the sum of dimension sizes:
    // sum((d.upper - d.lower + 1) for d in dimensions)
    Element    elements[];
};

struct Dimension {
    // Upper dimension bound, inclusive,
    // number of elements in the dimension
    // relative to the lower bound.
    int32      upper;

    // Lower dimension bound, always 1.
    int32      lower;
};

struct Element {
    // Encoded element data length in bytes.
    int32      length;

    // Element data.
    uint8     data[length];
};
```

Note: zero-length arrays (and sets) are represented as a 12-byte value where `dims` equal to zero regardless of the shape in type descriptor.

Sets of arrays are a special case. Every array within a set is wrapped in an Envelope. The full structure follows:

```
struct SetOfArrayValue {
    // Number of dimensions, currently must
    // always be 0 or 1. 0 indicates an empty set.
    int32 ndims;

    // Reserved.
    int32 reserved0;
```

(continues on next page)

(continued from previous page)

```

// Reserved.
int32 reserved1;

// Dimension data. Same layout as above.
Dimension dimensions[ndims];

// Envelope data, the number of elements
// in this array is the sum of dimension sizes:
// sum((d.upper - d.lower + 1) for d in dimensions)
Envelope elements[];
};

struct Envelope {
    // Encoded envelope element length in bytes.
    int32 length;

    // Number of elements, currently must
    // always be 1.
    int32 nelems;

    // Reserved.
    int32 reserved

    // Element data. Same layout as above.
    Element element[nelems];
};

```

#### 8.13.4.2 tuple<>, namedtuple<>, and object<>

The values are represented as the following structure:

```

struct TupleOrNamedTupleOrObjectValue {
    // Number of elements
    int32      nelems;

    // Element data.
    Element    elements[nelems];
};

struct Element {
    // Reserved.
    int32      reserved;

    // Encoded element data length in bytes.
    int32      length;

    // Element data.
    uint8      data[length];
};

```

Note that for objects, `Element.length` can be set to `-1`, which means an empty set.

### 8.13.4.3 Sparse Objects

The values are represented as the following structure:

```
struct SparseObjectValue {
    // Number of elements
    int32      nelems;

    // Element data.
    Element    elements[nelems];
};

struct Element {
    // Index of the element in the input shape.
    int32      index;

    // Encoded element data length in bytes.
    int32      length;

    // Element data.
    uint8      data[length];
};
```

### 8.13.4.4 Ranges

The ranges are represented as the following structure:

```
struct Range {
    // A bit mask of range definition.
    uint8<RangeFlag> flags;

    // Lower boundary data.
    Boundary    lower;

    // Upper boundary data.
    Boundary    upper;
};

struct Boundary {
    // Encoded boundary data length in bytes.
    int32      length;

    // Boundary data.
    uint8      data[length];
};

enum RangeFlag {
    // Empty range.
    EMPTY     = 0x0001;

    // Included lower boundary.
    LB_INC   = 0x0002;
};
```

(continues on next page)

(continued from previous page)

```
// Included upper boundary.  
UB_INC = 0x0004;  
  
// Infinity (excluded) lower boundary.  
LB_INF = 0x0008;  
  
// Infinity (excluded) upper boundary.  
UB_INF = 0x0010;  
};
```

#### 8.13.4.5 std::uuid

The `std::uuid` values are represented as a sequence of 16 unsigned byte values.

For example, the UUID value b9545c35-1fe7-485f-a6ea-f8ead251abd3 is represented as:

```
0xb9 0x54 0x5c 0x35 0x1f 0xe7 0x48 0x5f  
0xa6 0xea 0xf8 0xea 0xd2 0x51 0xab 0xd3
```

#### 8.13.4.6 std::str

The `std::str` values are represented as a UTF-8 encoded byte string. For example, the `str` value 'Hello! ' is encoded as:

```
0x48 0x65 0x6c 0x6c 0x6f 0x21 0x20 0xf0 0x9f 0x99 0x82
```

#### 8.13.4.7 std::bytes

The `std::bytes` values are represented as-is.

#### 8.13.4.8 std::int16

The `std::int16` values are represented as two bytes, most significant byte first.

For example, the `int16` value 6556 is represented as:

```
0x19 0x9c
```

#### 8.13.4.9 std::int32

The `std::int32` values are represented as four bytes, most significant byte first.

For example, the `int32` value 655665 is represented as:

```
0x00 0x0a 0x01 0x31
```

#### 8.13.4.10 std::int64

The `std::int64` values are represented as eight bytes, most significant byte first.

For example, the `int64` value 123456789987654321 is represented as:

```
0x01 0xb6 0x9b 0x4b 0xe0 0x52 0xfa 0xb1
```

#### 8.13.4.11 std::float32

The `std::float32` values are represented as a IEEE 754-2008 binary 32-bit value, most significant byte first.

For example, the `float32` value -15.625 is represented as:

```
0xc1 0x7a 0x00 0x00
```

#### 8.13.4.12 std::float64

The `std::float32` values are represented as a IEEE 754-2008 binary 64-bit value, most significant byte first.

For example, the `float64` value -15.625 is represented as:

```
0xc0 0x2f 0x40 0x00 0x00 0x00 0x00 0x00
```

#### 8.13.4.13 std::decimal

The `std::decimal` values are represented as the following structure:

```
struct Decimal {
    // Number of digits in digits[], can be 0.
    uint16          ndigits;

    // Weight of first digit.
    int16           weight;

    // Sign of the value
    uint16<DecimalSign> sign;

    // Value display scale.
    uint16          dscale;

    // base-10000 digits.
    uint16          digits[ndigits];
};

enum DecimalSign {
    // Positive value.
    POS      = 0x0000;

    // Negative value.
    NEG      = 0x4000;
};
```

The decimal values are represented as a sequence of base-10000 *digits*. The first digit is assumed to be multiplied by  $weight * 10000$ , i.e. there might be up to  $weight + 1$  digits before the decimal point. Trailing zeros can be absent. It is possible to have negative weight.

*dscale*, or display scale, is the nominal precision expressed as number of base-10 digits after the decimal point. It is always non-negative. *dscale* may be more than the number of physically present fractional digits, implying significant trailing zeroes. The actual number of digits physically present in the *digits* array contains trailing zeros to the next 4-byte increment (meaning that integer and fractional part are always distinct base-10000 digits).

For example, the decimal value `-15000.6250000` is represented as:

```
// ndigits
0x00 0x04

// weight
0x00 0x01

// sign
0x40 0x00

// dscale
0x00 0x07

// digits
0x00 0x01 0x13 0x88 0x18 0x6a 0x00 0x00
```

#### 8.13.4.14 std::bool

The `std::bool` values are represented as an `int8` with only two valid values: `0x01` for `true` and `0x00` for `false`.

#### 8.13.4.15 std::datetime

The `std::datetime` values are represented as a 64-bit integer, most significant byte first. The value is the number of *microseconds* between the encoded datetime and January 1st 2000, 00:00 UTC. A Unix timestamp can be converted into an EdgeDB `datetime` value using this formula:

```
edb_datetime = (unix_ts + 946684800) * 1000000
```

For example, the `datetime` value '`2019-05-06T12:00+00:00`' is encoded as:

```
0x00 0x02 0x2b 0x35 0x9b 0xc4 0x10 0x00
```

See [client libraries](#) section for more info about how to handle different precision when encoding data.

#### 8.13.4.16 `cal::local_datetime`

The `cal::local_datetime` values are represented as a 64-bit integer, most significant byte first. The value is the number of *microseconds* between the encoded datetime and January 1st 2000, 00:00.

For example, the `local_datetime` value '`2019-05-06T12:00`' is encoded as:

```
0x00 0x02 0x2b 0x35 0x9b 0xc4 0x10 0x00
```

See *client libraries* section for more info about how to handle different precision when encoding data.

#### 8.13.4.17 `cal::local_date`

The `cal::local_date` values are represented as a 32-bit integer, most significant byte first. The value is the number of *days* between the encoded date and January 1st 2000.

For example, the `local_date` value '`2019-05-06`' is encoded as:

```
0x00 0x00 0x1b 0x99
```

#### 8.13.4.18 `cal::local_time`

The `cal::local_time` values are represented as a 64-bit integer, most significant byte first. The value is the number of *microseconds* since midnight.

For example, the `local_time` value '`12:10`' is encoded as:

```
0x00 0x00 0x00 0x0a 0x32 0xae 0xf6 0x00
```

See *client libraries* section for more info about how to handle different precision when encoding data.

#### 8.13.4.19 `std::duration`

The `std::duration` values are represented as the following structure:

```
struct Duration {
    int64    microseconds;

    // deprecated, is always 0
    int32    days;

    // deprecated, is always 0
    int32    months;
};
```

For example, the duration value '`48 hours 45 minutes 7.6 seconds`' is encoded as:

```
// microseconds
0x00 0x00 0x00 0x28 0xdd 0x11 0x72 0x80

// days
0x00 0x00 0x00 0x00
```

(continues on next page)

(continued from previous page)

```
// months
0x00 0x00 0x00 0x00
```

See [client libraries](#) section for more info about how to handle different precision when encoding data.

#### 8.13.4.20 cal::relative\_duration

The `cal::relative_duration` values are represented as the following structure:

```
struct Duration {
    int64  microseconds;
    int32  days;
    int32  months;
};
```

For example, the `cal::relative_duration` value '2 years 7 months 16 days 48 hours 45 minutes 7.6 seconds' is encoded as:

```
// microseconds
0x00 0x00 0x00 0x28 0xdd 0x11 0x72 0x80

// days
0x00 0x00 0x00 0x10

// months
0x00 0x00 0x00 0x1f
```

See [client libraries](#) section for more info about how to handle different precision when encoding data.

#### 8.13.4.21 cal::date\_duration

The `cal::date_duration` values are represented as the following structure:

```
struct DateDuration {
    int64  reserved;
    int32  days;
    int32  months;
};
```

For example, the `cal::date_duration` value '1 years 2 days' is encoded as:

```
// reserved
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

// days
0x00 0x00 0x00 0x02

// months
0x00 0x00 0x00 0x0c
```

### 8.13.4.22 std::json

The `std::json` values are represented as the following structure:

```
struct JSON {
    uint8  format;
    uint8  jsondata[];
};
```

`format` is currently always 1, and `jsondata` is a UTF-8 encoded JSON string.

### 8.13.4.23 std::bigint

The `std::bigint` values are represented as the following structure:

```
struct BigInt {
    // Number of digits in digits[], can be 0.
    uint16          ndigits;

    // Weight of first digit.
    int16           weight;

    // Sign of the value
    uint16<DecimalSign> sign;

    // Reserved value, must be zero
    uint16          reserved;

    // base-10000 digits.
    uint16          digits[ndigits];
};

enum BigIntSign {
    // Positive value.
    POS      = 0x0000;

    // Negative value.
    NEG      = 0x4000;
};
```

The decimal values are represented as a sequence of base-10000 `digits`. The first digit is assumed to be multiplied by `weight * 10000`, i.e. there might be up to `weight + 1` digits. Trailing zeros can be absent.

For example, the bigint value `-15000` is represented as:

```
// ndigits
0x00 0x02

// weight
0x00 0x01

// sign
0x40 0x00
```

(continues on next page)

(continued from previous page)

```
// reserved  
0x00 0x00  
  
// digits  
0x00 0x01 0x13 0x88
```

#### 8.13.4.24 cfg::memory

The `cfg::memory` values are represented as a number of *bytes* encoded as a 64-bit integer, most significant byte first.

For example, the `cfg::memory` value 123MiB is represented as:

```
0x00 0x00 0x00 0x00 0x07 0xb0 0x00 0x00
```

### 8.13.5 Conventions and data Types

The message format descriptions in this section use a C-like struct definitions to describe their layout. The structs are *packed*, i.e. there are never any alignment gaps.

The following data types are used in the descriptions:

<code>int8</code>	8-bit integer
<code>int16</code>	16-bit integer, most significant byte first
<code>int32</code>	32-bit integer, most significant byte first
<code>int64</code>	64-bit integer, most significant byte first
<code>uint8</code>	8-bit unsigned integer
<code>uint16</code>	16-bit unsigned integer, most significant byte first
<code>uint32</code>	32-bit unsigned integer, most significant byte first
<code>uint64</code>	64-bit unsigned integer, most significant byte first
<code>int8&lt;T&gt;</code> or <code>uint8&lt;T&gt;</code>	an 8-bit signed or unsigned integer enumeration, where <code>T</code> denotes the name of the enumeration
<code>string</code>	a UTF-8 encoded text string prefixed with its byte length as <code>uint32</code>
<code>bytes</code>	a byte string prefixed with its length as <code>uint32</code>
<code>KeyValue</code>	<pre><code>struct KeyValue {     // Key code (specific to the type of the     // Message).     uint16      code;      // Value data.     bytes       value; };</code></pre>
<code>Annotation</code>	<pre><code>struct Annotation {     // Name of the annotation     string      name;      // Value of the annotation (in JSON     // format).     string      value; };</code></pre>
<code>uuid</code>	an array of 16 bytes with no length prefix, equivalent to <code>byte[16]</code>

### 8.13.6 Message Format

All messages in the EdgeDB wire protocol have the following format:

```
struct {
    uint8   message_type;
    int32   payload_length;
    uint8   payload[payload_length - 4];
};
```

The server and the client *MUST* not fragment messages. I.e the complete message must be sent before starting a new message. It's advised that whole message should be buffered before initiating a network call (but this requirement is neither observable nor enforceable at the other side). It's also common to buffer the whole message on the receiver side before starting to process it.

### 8.13.7 Errors

At any point the server may send an *ErrorResponse* indicating an error condition. This is implied in the message flow documentation, and only successful paths are explicitly documented. The handling of the *ErrorResponse* message depends on the connection phase, as well as the severity of the error.

If the server is not able to recover from an error, the connection is closed immediately after an *ErrorResponse* message is sent.

### 8.13.8 Logs

Similarly to *ErrorResponse* the server may send a *LogMessage* message. The client should handle the message and continue as before.

### 8.13.9 Message Flow

There are two main phases in the lifetime of an EdgeDB connection: the connection phase, and the command phase. The connection phase is responsible for negotiating the protocol and connection parameters, including authentication. The command phase is the regular operation phase where the server is processing queries sent by the client.

#### 8.13.9.1 Connection Phase

To begin a session, a client opens a connection to the server, and sends the *ClientHandshake*. The server responds in one of three ways:

1. One of the authentication messages (see *below*);
2. *ServerHandshake* followed by one of the authentication messages;
3. *ErrorResponse* which indicates an invalid client handshake message.

*ServerHandshake* is only sent if the requested connection parameters cannot be fully satisfied; the server responds to offer the protocol parameters it is willing to support. Client may proceed by noting lower protocol version and/or absent extensions. Client *MUST* close the connection if protocol version is unsupported. Server *MUST* send subset of the extensions received in *ClientHandshake* (i.e. it never adds extra ones).

While it's not required by the protocol specification itself, EdgeDB server currently requires setting the following params in *ClientHandshake*:

- **user** – username for authentication
- **database** – database to connect to

#### 8.13.9.2 Authentication

The server then initiates the authentication cycle by sending an authentication request message, to which the client must respond with an appropriate authentication response message.

The following messages are sent by the server in the authentication cycle:

***AuthenticationOK*** Authentication is successful.

***AuthenticationSASL*** The client must now initiate a SASL negotiation, using one of the SASL mechanisms listed in the message. The client will send an *AuthenticationSASLInitialResponse* with the name of the selected mechanism, and the first part of the SASL data stream in response to this. If further messages are needed, the server will respond with *AuthenticationSASLContinue*.

***AuthenticationSASLContinue*** This message contains challenge data from the previous step of SASL negotiation (*AuthenticationSASL*, or a previous *AuthenticationSASLContinue*). The client must respond with an *AuthenticationSASLResponse* message.

***AuthenticationSASLFinal*** SASL authentication has completed with additional mechanism-specific data for the client. The server will next send *AuthenticationOK* to indicate successful authentication, or an *ErrorResponse* to indicate failure. This message is sent only if the SASL mechanism specifies additional data to be sent from server to client at completion.

If the frontend does not support the authentication method requested by the server, then it should immediately close the connection.

Once the server has confirmed successful authentication with *AuthenticationOK*, it then sends one or more of the following messages:

***ServerKeyData*** This message provides per-connection secret-key data that the client must save if it wants to be able to issue certain requests later. The client should not respond to this message.

***ParameterStatus*** This message informs the frontend about the setting of certain server parameters. The client can ignore this message, or record the settings for its future use. The client should not respond to this message.

The connection phase ends when the server sends the first *ReadyForCommand* message, indicating the start of a command cycle.

### 8.13.9.3 Command Phase

In the command phase, the server expects the client to send one of the following messages:

***Parse*** Instructs the server to parse the provided command or commands for execution. The server responds with a *CommandDataDescription* containing the *type descriptor* data necessary to perform data I/O for this command.

***Execute*** Execute the provided command or commands. This message expects the client to declare a correct *type descriptor* identifier for command arguments. If the declared input type descriptor does not match the expected value, a *CommandDataDescription* message is returned followed by a *ParameterTypeMismatchError* in an *ErrorResponse* message.

If the declared output type descriptor does not match, the server will send a *CommandDataDescription* prior to sending any *Data* messages.

The client could attach state data in both messages. When doing so, the client must also set a correct *type descriptor* identifier for the state data. If the declared state type descriptor does not match the expected value, a *StateDataDescription* message is returned followed by a *StateMismatchError* in an *ErrorResponse* message. However, the special type id of zero `00000000-0000-0000-0000-000000000000` for empty/default state is always a match.

Each of the messages could contain one or more EdgeQL commands separated by a semicolon (;). If more than one EdgeQL command is found in a single message, the server will treat the commands as an EdgeQL script. EdgeQL scripts are always atomic, they will be executed in an implicit transaction block if no explicit transaction is currently active. Therefore, EdgeQL scripts have limitations on the kinds of EdgeQL commands they can contain:

- Transaction control commands are not allowed, like `start transaction`, `commit`, `declare savepoint`, or `rollback to savepoint`.
- Non-transactional commands, like `create database` or `configure instance` are not allowed.

In the command phase, the server can be in one of the three main states:

- *idle*: server is waiting for a command;
- *busy*: server is executing a command;
- *error*: server encountered an error and is discarding incoming messages.

Whenever a server switches to the *idle* state, it sends a *ReadyForCommand* message.

Whenever a server encounters an error, it sends an *ErrorResponse* message and switches into the *error* state.

To switch a server from the *error* state into the *idle* state, a *Sync* message must be sent by the client.

#### 8.13.9.4 Dump Database Flow

Backup flow goes as following:

1. Client sends *Dump* message
2. Server sends *Dump Header* message
3. Server sends one or more *Dump Block* messages
4. Server sends *CommandComplete* message

Usually client should send *Sync* after Dump message to finish implicit transaction.

#### 8.13.9.5 Restore Database Flow

Restore procedure fills up the database the client is connected to with the schema and data from the dump file.

Flow is the following:

1. Client sends *Restore* message with the dump header block
2. Server sends *RestoreReady* message as a confirmation that it has accepted the header, restored schema and ready to receive data blocks
3. Clients sends one or more *RestoreBlock* messages
4. Client sends *RestoreEof* message
5. Server sends *CommandComplete* message

Note: *ErrorResponse* may be sent from the server at any time. In case of error, *Sync* must be sent and all subsequent messages ignored until *ReadyForCommand* is received.

Restore protocol doesn't require a *Sync* message except for error cases.

#### 8.13.10 Termination

The normal termination procedure is that the client sends a *Terminate* message and immediately closes the connection. On receipt of this message, the server cleans up the connection resources and closes the connection.

In some cases the server might disconnect without a client request to do so. In such cases the server will attempt to send an *ErrorResponse* or a *LogMessage* message to indicate the reason for the disconnection.

## 8.14 Client Libraries

EdgeDB client libraries are a bit higher level than usual database bindings. In particular, they contain:

- Structured data retrieval
- Connection pooling
- Retrying of failed transactions and queries

Additionally, client libraries might provide:

- Code generation for type-safe database access
- Query builder

This is a **work-in-progress** reference for writing client libraries for EdgeDB.

External Links:

- [Official Client Libraries](#)
- [Binary protocol](#)
- [RFC 1004 - Robust Client API](#)

Contents:

### 8.14.1 Date/Time Handling

EdgeDB has 6 types related to date and time handling:

- [`datetime` \(\*binary format\*\)](#)
- [`duration` \(\*binary format\*\)](#)
- [`cal::local\_datetime` \(\*binary format\*\)](#)
- [`cal::local\_date` \(\*binary format\*\)](#)
- [`cal::relative\_duration` \(\*binary format\*\)](#)
- [`cal::date\_duration` \(\*binary format\*\)](#)

Usually we try to map those types to the respective language-native types, with the following caveats:

- The type in standard library
- It has enough range (EdgeDB has timestamps from year 1 to 9999)
- And it has good enough precision (at least microseconds)

If any of the above criteria is not met, we usually provide a custom type in the client library itself that can be converted to a type from the language's standard library or from a popular third-party library. Exception: The JavaScript `Date` type (which is actually a timestamp) has millisecond precision. We decided it would be better to use that type by default even though it doesn't have sufficient precision.

### 8.14.1.1 Precision

`datetime`, `duration`, `cal::local_datetime` and `cal::relative_duration` all have precision of **1 millisecond**.

This means that if language-native type have a bigger precision such as nanosecond, client library has to round that timestamp when encoding it for EdgeDB.

We use **rounding to the nearest even** for that operation. Here are some examples of timestamps with high precision, and how they are stored in the database:

```
2022-02-24T05:43:03.123456789Z → 2022-02-24T05:43:03.123457Z  
  
2022-02-24T05:43:03.000002345Z → 2022-02-24T05:43:03.000002Z  
2022-02-24T05:43:03.000002500Z → 2022-02-24T05:43:03.000002Z  
2022-02-24T05:43:03.000002501Z → 2022-02-24T05:43:03.000003Z  
2022-02-24T05:43:03.000002499Z → 2022-02-24T05:43:03.000002Z  
  
2022-02-24T05:43:03.000001234Z → 2022-02-24T05:43:03.000001Z  
2022-02-24T05:43:03.000001500Z → 2022-02-24T05:43:03.000002Z  
2022-02-24T05:43:03.000001501Z → 2022-02-24T05:43:03.000002Z  
2022-02-24T05:43:03.000001499Z → 2022-02-24T05:43:03.000001Z
```

---

**Note:** A quick refresher on rounding types: If we perform multiple operations of summing while rounding half-up or rounding half-down, the error margin of the resulting value tends to increase. If we round half-to-even instead, the expected value of summing tends to be more accurate.

---

Note as described in [datetime protocol documentation](#) the value is encoded as a *signed* microseconds delta since a fixed time. Some care must be taken when rounding negative microsecond values. See [tests for Rust implementation](#) for a good set of test cases.

Rounding to the nearest even applies to all operations that client libraries perform, in particular:

1. Encoding timestamps *and* time deltas (see the [list of types](#)) to the binary format if precision of the native type is higher than microseconds.
2. Decoding timestamps *and* time deltas from the binary format if precision of native type is lower than microseconds (applies for JavaScript for example)
3. Converting from EdgeDB specific type (if there is one) to native type and back (depending on the difference in precision)
4. Parsing a string to an EdgeDB specific type (this operation is optional to implement, but if it is implemented, it must obey the rules)

## 8.15 Administration

Administrative commands for managing EdgeDB:

- `configure`  
Configure server behavior.
- `database`  
Create or remove a database.

- *role*

Create, remove or alter a role.

## 8.15.1 Configure

### eql-statement

`configure` – change a server configuration parameter

```
configure {session | current database | instance}
    set <parameter> := <value> ;
configure instance insert <parameter-class> <insert-shape> ;
configure {session | current database | instance} reset <parameter> ;
configure {current database | instance}
    reset <parameter-class> [ filter <filter-expr> ] ;
```

### 8.15.1.1 Description

This command allows altering the server configuration.

The effects of `configure session` last until the end of the current session. Some configuration parameters cannot be modified by `configure session` and can only be set by `configure instance`.

`configure current database` is used to configure an individual EdgeDB database within a server instance with the changes persisted across server restarts.

`configure instance` is used to configure the entire EdgeDB instance with the changes persisted across server restarts. This variant acts directly on the file system and cannot be rolled back, so it cannot be used in a transaction block.

The `configure instance insert` variant is used for composite configuration parameters, such as `Auth`.

### 8.15.1.2 Parameters

**<parameter>** The name of a primitive configuration parameter. Available configuration parameters are described in the [Config](#) section.

**<parameter-class>** The name of a composite configuration value class. Available configuration classes are described in the [Config](#) section.

**<filter-expr>** An expression that returns a value of type `std::bool`. Only configuration objects matching this condition will be affected.

### 8.15.1.3 Examples

Set the `listen_addresses` parameter:

```
configure instance set listen_addresses := {'127.0.0.1', '::1'};
```

Set the `query_work_mem` parameter for the duration of the session:

```
configure session set query_work_mem := <cfg::memory>'4MiB';
```

Set the same parameter, but for the current database:

```
configure current database set query_work_mem := <cfg::memory>'4MiB';
```

Add a Trust authentication method for “my\_user”:

```
configure instance insert Auth {  
    priority := 1,  
    method := (insert Trust),  
    user := 'my_user'  
};
```

Remove all Trust authentication methods:

```
configure instance reset Auth filter Auth.method is Trust;
```

## 8.15.2 Database

### edb-alt-title Databases

This section describes the administrative commands pertaining to *databases*.

#### 8.15.2.1 Create database

##### eql-statement

Create a new database.

```
create database <name> ;
```

##### 8.15.2.1.1 Description

The command `create database` creates a new EdgeDB database.

The new database will be created with all standard schemas prepopulated.

##### 8.15.2.1.2 Examples

Create a new database:

```
create database appdb;
```

#### 8.15.2.2 Drop database

##### eql-statement

Remove a database.

```
drop database <name> ;
```

### 8.15.2.2.1 Description

The command `drop database` removes an existing database. It cannot be executed while there are existing connections to the target database.

**Warning:** Executing `drop database` removes data permanently and cannot be undone.

### 8.15.2.2.2 Examples

Remove a database:

```
drop database appdb;
```

## 8.15.3 Role

### edb-alt-title Roles

This section describes the administrative commands pertaining to *roles*.

### 8.15.3.1 Create role

#### eql-statement

Create a role.

```
create superuser role <name> [ extending <base> [, ...] ]
"{"<subcommand>; [...]"}";
# where <subcommand> is one of
set password := <password>
```

### 8.15.3.1.1 Description

The command `create role` defines a new database role.

**superuser** If specified, the created role will have the *superuser* status, and will be exempt from all permission checks. Currently, the `superuser` qualifier is mandatory, i.e. it is not possible to create non-superuser roles for now.

**<name>** The name of the role to create.

**extending <base> [, ...]** If specified, declares the parent roles for this role. The role inherits all the privileges of the parents.

The following subcommands are allowed in the `create role` block:

**set password := <password>** Set the password for the role.

### 8.15.3.1.2 Examples

Create a new role:

```
create role alice {  
    set password := 'wonderland';  
};
```

### 8.15.3.2 Alter role

#### eql-statement

Alter an existing role.

```
alter role <name> "{" <subcommand>; [...] "}" ;  
  
# where <subcommand> is one of  
  
    rename to <newname>  
    set password := <password>  
    extending ...
```

#### 8.15.3.2.1 Description

The command `alter role` changes the settings of an existing role.

**<name>** The name of the role to alter.

The following subcommands are allowed in the `alter role` block:

**rename to <newname>** Change the name of the role to *newname*.

**extending ...** Alter the role parent list. The full syntax of this subcommand is:

```
extending <name> [, ...]  
[ first | last | before <parent> | after <parent> ]
```

This subcommand makes the role a child of the specified list of parent roles. The role inherits all the privileges of the parents.

It is possible to specify the position in the parent list using the following optional keywords:

- **first** – insert parent(s) at the beginning of the parent list,
- **last** – insert parent(s) at the end of the parent list,
- **before <parent>** – insert parent(s) before an existing *parent*,
- **after <parent>** – insert parent(s) after an existing *parent*.

### 8.15.3.2.2 Examples

Alter a role:

```
alter role alice {  
    set password := 'new password';  
};
```

### 8.15.3.3 Drop role

#### eql-statement

Remove a role.

```
drop role <name> ;
```

#### 8.15.3.3.1 Description

The command `drop role` removes an existing role.

### 8.15.3.3.2 Examples

Remove a role:

```
drop role alice;
```

This section contains comprehensive reference documentation on the internals of EdgeDB, the binary protocol, the formal syntax of EdgeQL, and more.



## CHANGELOG

Changes introduced in all of the releases of EdgeDB so far:

### 9.1 v1.0

**edb-alt-title** EdgeDB v1 (Nova)



EdgeDB 1.0 was released on February 10, 2022. Read the announcement blog post [here](#).

We would like to thank our community for reporting issues and contributing fixes. You are awesome!

#### 9.1.1 1.4

- Avoid unnecessary updates to parent views and triggers ([#3771](#))
- Drop broken special case for nested insert FOR ([#3797](#))
- Fix IN array\_unpack for bigints ([#3820](#))
- Put more parens around index expressions in generated DDL ([#3822](#))
- Fix a weird computable/alias interaction ([#3828](#))
- Support linkprops on backlinks ([#3841](#))
- Fix generation of dummy\_pathid in nonconflict ctes ([#3848](#))

- Always correctly handle variadic arguments when producing AST from migrations (#3855)

### 9.1.2 1.3

- Fix a multiplicity computation bug for tuples (#3632`)
- Fix a multiplicity issue with doubly nested loops (#3636)
- Fix a bug involving computable shadowing (#3652)
- Make replace\_prefix change PointerRefs to make types line up (#3642)
- Search the source\_rvar for materialized refs (#3657)
- Fix min/max when no source aspect is present (#3664)
- Don't create UnionTypeShell for views of unions (#3670)
- Fix a collection of json casting bugs (#3676)
- ha/stolon: Don't attempt to parse unsuccessful Consul responses (#3698)
- ha/stolon: Add exponential backoff on unsuccessful Consul KV responses (#3699)
- Make sure the sets in conflict clauses are have correct scope (#3686)
- Produce an error instead of malformed SQL on `insert foo { name }` (#3687)
- Don't treat everything with a binding as STABLE (#3689)
- Check that index expressions are immutable (#3690)
- Fix fetching computed property of UNION of same type (#3691)
- Fix references to `__subject__` in object constraints (#3695)
- Fix unions of DML overlays on reverse inline pointers (#3694)
- Fix some inheritance issues with renames and expr refs (#3696)
- Fix a card inference bug when dealing with computables (#3628)
- Fix a collection of nested shape path reference issues (#3700)
- Produce a proper error on `describe` of nonexisting function, module (#3701)
- Pin the version of edgedb-cli that we use (#3705)
- Fix [NOT] IN `array_unpack(foo)` when foo is empty (#3752)
- Inject SYNC between state restore & START TRANSACTION in [execute] flow. (#3749`)
- Fail if local Postgres cluster fails to start

### 9.1.3 1.2

- Add on-demand compiler pool scaling (#3550).

Use `--compiler-pool-mode=on_demand` to switch to the new mode, which will spawn new compiler worker process every 3 seconds if the compiling requests keep queueing up. The upper limit on the number of these spawned processes is the number of CPUs. After 60 seconds without compiling requests, the compiler pool will scale down to `--compiler-pool-size` with a default of 1 under on-demand mode.

- Fix an issue with default module and module aliases inside transactions (#3604).
- Add `reset on target delete` to `DDL` in order to fix some migration bugs concerning links (#3611).

- Enforce newly created *exclusive* constraints across existing data (#3613).
- Fix an issue with a self-referencing *update* (#3605).
- Fix an issue with a constraint bug in *update* (#3603).
- Fix a constraint bug in *update* (#3603).
- Fix an SDL issue with computed links referencing each other (#3499).
- Fix self-referencing nested mutations in GraphQL (#3470).
- Fix GraphQL fragments for types (#3514).
- Add *in* operator to GraphQL (#3443).
- Fix cardinality inference in some special cases (#3590).
- Fix inheritance from enum types (#3578).
- Fix cardinality inference bug and extend cardinality restrictions to longer paths (#3566).

Specifically, correctly infer that filtering on a long path where each hop is *exclusive* produces at most one result.

For example: `select Foo filter .bar.baz = 'key'` should have cardinality of at most one if both `bar` and `baz` have *exclusive* constraints.

- Fix issues involving scalar set identity (#3525).
- Fix exclusive constraints on tuple properties (#3559).

#### 9.1.4 1.1

- Fix a migration issue with handling `default` on inherited *links* or *properties* (#3544).
- Fix a migration issue with an inherited *property* (#3542).
- Fix a migration issue with dropping a type (#3521).
- Fix a migration issue with changing a *link* from `single` to `multi` (#3392).
- Provide a more detailed message for *constraints* errors (#3522).
- Produce an error on an invalid regex (#3412).
- Produce a proper error when an expression is invalid in a certain special contexts, such as `default` (#3494).
- Format the database name correctly in `DuplicateDatabaseDefinitionError` (#3228)
- Fix an error when working with a composite *exclusive* constraint (#3502).
- Correctly infer cardinality of a property on a multi link in the context of the *constraint* on that link (#3536).
- Disable changing the concrete base of a *scalar type* (#3529).
- Avoid generating pointless self joins (#2567).
- Fix IPv6 address parsing (#3454).
- If a type has object instances, it cannot be made `abstract` (#3399).
- Fix an issue that sometimes caused *with* block variables to be unusable (#3385).

### 9.1.5 Pre-releases

EdgeDB 1.0 had a series of pre-preleases. Read the full history [here](#):

Alpha 2, alpha 3, alpha 4, alpha 5, alpha 6, alpha 7, beta 1, beta 2, beta 3, RC 1, RC 2, RC 3, RC 4, RC 5.

## 9.2 v2.0

**edb-alt-title** EdgeDB v2 (Sagittarius)



EdgeDB 2.0 was released on July 28th, 2022. Read the announcement blog post [here](#).

We would like to thank our community for reporting issues and contributing fixes. You are awesome!

To play with the new features, install [the CLI](#) and initialize a new project. For an interesting schema with test data, check out the [MCU Sandbox](#) repo.

```
$ edgedb project init
```

### 9.2.1 Upgrading

#### Local instances

To upgrade a local project, run the following command inside the project directory.

```
$ edgedb project upgrade --to-latest
```

Alternatively, specify an instance name if you aren't using a project.

```
$ edgedb project upgrade --to-latest -I my_instance
```

#### Hosted instances

To upgrade a remote (hosted) instance, we recommend the following dump-and-restore process.

1. Spin up an empty 2.0 instance by following one of our [deployment guides](#). These guides have been updated to 2.0. Keep the DSN of the newly created instance handy.

- Take your application offline, then dump your v1.x database with the CLI

```
$ edgedb dump --dsn <old dsn> --all my_database.dump/
```

This will dump the schema and contents of your current database to a file on your local disk called `my_database.dump`. The file name isn't important.

- Restore the empty v2.x instance from the dump

```
$ edgedb restore --all my_database.dump/ --dsn <new dsn>
```

Once the restore is complete, update your application to connect to the new instance.

This process will involve some downtime, specifically during steps 2 and 3. We are working on an in-place upgrade workflow that will reduce the amount of downtime involved and avoid the need to spin up a new instance. We'll publish that soon; join the Discord for updates. Though for most applications the dump-and-restore workflow will be simpler and less error-prone.

### 9.2.1.1 Client libraries

We've released new versions of our JavaScript and Python client libraries that support all 2.0 features and implement the updated protocol. These versions are backwards compatible with v1.x instances, so we encourage all users to upgrade.

<code>TypeScript/JS</code>	<code>edgedb@0.21.0</code>
<code>Python</code>	<code>edgedb@0.24.0</code>
<code>Golang</code>	<code>edgedb@0.12.0</code>
<code>Rust</code>	<code>edgedb-tokio@0.3.0</code>
<code>.NET (community-maintained)</code>	<code>EdgeDB.Net.Driver@0.3.0</code>
<code>Elixir (community-maintained)</code>	<code>edgedb@0.4.0</code>

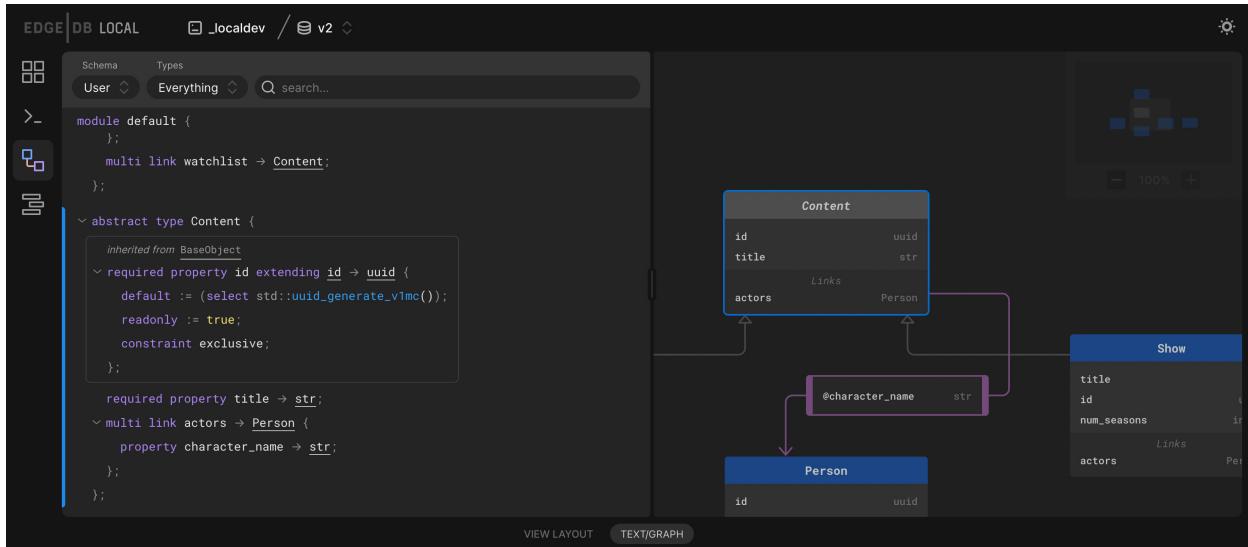
## 9.2.2 New features

### 9.2.2.1 Integrated admin UI

All v2 instances ship with a built-in rich admin GUI. Access it by running `edgedb ui` inside any *EdgeDB project*, or specify a local instance name with `edgedb ui -I my_inst`. The command opens the instance's admin UI using the default system browser.

The current iteration of the GUI has

- a data browser and editor
- a REPL for writing and executing EdgeQL queries
- a schema introspection tool with text-based and graphical visualizations of the instance's current schema



### 9.2.2.2 Analytical queries with GROUP

The new `GROUP` expression can be used to partition and aggregate data. The output of `GROUP` are *free objects* representing each group, including the grouping *key*, the grouping *key*, and the set of elements.

```
db> group Movie { title } by .release_year;
{
  {
    key: {release_year: 2017},
    grouping: {'release_year'},
    elements: {
      default::Movie {title: 'Guardians of the Galaxy Vol. 2'},
      default::Movie {title: 'Spider-Man: Homecoming'},
      default::Movie {title: 'Thor: Ragnarok'},
    },
  },
  {
    key: {release_year: 2013},
    grouping: {'release_year'},
    elements: {
      default::Movie {title: 'Iron Man 3'},
      default::Movie {title: 'Thor: The Dark World'},
    },
  },
  ...
}
```

Browse the [docs](#) for more details and examples, or refer to the original [RFC 1009](#).

### 9.2.2.3 Global variables

Your schema can now contain *global variables*. These are contextual variables that are provided by the client and can be referenced in your queries and schema.

```
global current_user -> uuid;

select User filter .id = global current_user;
```

Client libraries have been updated to provide method for attaching global variables to a `Client` instance; these values are sent along with all queries originating from that `Client`.

Listing 1: typescript

```
import {createClient} from 'edgedb';

const client = createClient().withGlobals({
    current_user: '2141a5b4-5634-4ccc-b835-437863534c51',
});

await client.query(`select global current_user;`);
```

Listing 2: python

```
from edgedb import create_client

client = create_client().with_globals({
    'current_user': '580cc652-8ab8-4a20-8db9-4c79a4b1fd81'
})

result = client.query("""
    select global current_user;
""")
```

Listing 3: go

```
package main

import (
    "context"
    "fmt"
    "log"

    "github.com/edgedb/edgedb-go"
)

func main() {
    ctx := context.Background()
    client, err := edgedb.CreateClient(ctx, edgedb.Options{})
    if err != nil {
        log.Fatal(err)
    }
    defer client.Close()
```

(continues on next page)

(continued from previous page)

```

id, err := edgedb.ParseUUID("2141a5b4-5634-4ccc-b835-437863534c51")
if err != nil {
    log.Fatal(err)
}

var result edgedb.UUID
err = client.
    WithGlobals(map[string]interface{}{"current_user": id}).
    QuerySingle(ctx, "SELECT global current_user;", &result)
if err != nil {
    log.Fatal(err)
}

fmt.Println(result)
}

```

Globals are primarily intended as an enabling mechanism for object-level security.

#### 9.2.2.4 Object-level security

Object types can now be augmented with object-level access policies. When combined with global variables, access policies can be used to push authorization logic into the database.

```

global current_user -> uuid;

type User {
    required property email -> str { constraint exclusive; };
}

type BlogPost {
    required property title -> str;
    link author -> User;
    access policy own_posts allow all using (
        .author.id ?= global current_user
    )
}

```

Refer to [the docs](#) or [RFC 1011](#) for full details.

#### 9.2.2.5 Range types

EdgeDB now supports *range types* representing intervals of values.

```

db> select range(1, 10);
{range(1, 10, inc_lower := true, inc_upper := false)}
db> select range_unpack(range(1, 10))
{1, 2, 3, 4, 5, 6, 7, 8, 9}

```

### 9.2.2.6 The `cal::date_duration` type

This release also introduces a new datatype `cal::date_duration` to represent a span of *months/days*. It is nearly equivalent to the existing `cal::relative_duration` but cannot represent sub-day durations.

This type is primarily intended to simplify `cal::local_date` logic.

```
db> select <cal::local_date>'2022-06-25' +
...   <cal::date_duration>'5 days';
{<cal::local_date>'2022-06-30'}
db> select <cal::local_date>'2022-06-30' -
...   <cal::local_date>'2022-06-25';
{<cal::date_duration>'P5D'}
```

### 9.2.2.7 Source deletion policies

Add deletion cascade functionality with `on source delete`.

```
type BlogPost {
    property title -> str;
}

type Person {
    multi link posts -> BlogPost {
        on source delete delete target;
    }
}
```

Under this policy, deleting a User will unconditionally delete its `posts` as well.

To avoid deleting a Post that is linked to by other schema entities, append `if orphan`.

```
type Person {
    multi link posts -> BlogPost {
-        on source delete delete target;
+        on source delete delete target if orphan;
    }
}
```

## 9.2.3 Additional changes

### 9.2.3.1 EdgeQL

- Support additional operations on local date and time types, including `duration_get()`, `cal::duration_normalize_hours()`, and `cal::duration_normalize_days()`. Per RFC 1013.
- Support user-provided values for the `id` property when inserting objects (#3895). This can be useful when migrating data from an existing database.

```
insert User {
    id := <uuid>"5abf67cc-9f9f-4bbc-b009-d117d463a12e",
    email := "jayz@example.com"
}
```

- Support partial constraints and indexes (#3949, [docs](#)).
- Add the new `json_set()` function (#4118).

### 9.2.3.2 Server

- Support socket activation to reduce memory footprint on developer machines (#3899).
- Introduce edgedb+http, a which tunnels the binary protocol over HTTP using JWT for authentication (#3979).
- Support using JWT to authenticate to local instances (#3991).

### 9.2.3.3 Bug fixes

- Generate unique `id` fields for each free shape object, and don't use an actual in-database object to represent it, and make multiplicity inference understand free shapes better (#3631, #3633, #3634).
- Fail if local Postgres cluster fails to start.
- Add `cfg::memory` to base types descriptor IDs table (#3882).
- Fix a cross-type exclusive constraint bug that could allow exclusive constraints to be violated in some complex type hierarchies (#3887).
- Fix issue where server might attempt to acquire one more connection than it is configured to permit (#3901).
- Fix use of `assert_exists` on properties that are being directly output (#3911).
- Fix a scope leakage that could cause a link referenced inside a computable to improperly correlate with something outside the computable (#3912).
- Fix a number of issues with the floordiv (`//`) and modulus (`%`) operators where we could return incorrect values or produce spurious errors, especially on very large values (#3909).
- Allow adding annotations to `abstract annotation` definitions (#3929).
- Expose `body` and `language` fields on `schema::Function` (#3944).
- Make indexes extend from `schema::InheritingObject` (#3942).
- Fix some mis-compilations of nested shapes inside calls to functions like `assert_single` (#3927).
- Fix SET TYPE on properties with default values (#3954).
- Fix `describe/populate/describe` sequence (#3959).
- Upgrade many casts and functions from “Stable” to “Immutable” (#3975).
- Fix link properties in type filtered shape links (#3987).
- Allow DML statements in free shapes (#4002).
- Allow customizing assertion messages in `assert_exists` and friends (#4019).

### 9.2.3.4 Protocol overhaul

- A new version of the protocol—version 1.0—has been introduced. It eliminates all server state associated with connections that do not use transactions.
- Support passing parameters to and returning values from multi-statement scripts.

### 9.2.4 2.1

- Fix global defaults with nontrivial computation (#4182)
- Fix migration that removes policy using clause (#4183)
- Support ELSE-less UNLESS CONFLICT on explicit id INSERT (#4185)
- Don't create constraints on derived views when adding a pointer to a type (#4187)
- Fix a bunch of missing source contexts in declarative (#4188)
- Fix an ISE when a computed link is directly a property reference (#4193)
- Fix an ISE when using an empty shape in some contexts (#4194)
- Fix a number of error messages involving collection types in schemas (#4195)
- Avoid doing semi-joins after a sequence of single links (#4196)
- Make range() properly strict in its non-optional arguments (#4207)
- Allow multiple FDs per socket in activation (#4189)
- Add SCRAM authentication over HTTP (#4197)
- Always arm auto-shutdown timer when it's greater than zero (#4214)
- Fix json -> array<json> cast of '[]' (#4217)

### 9.2.5 2.2

- Support UNLESS CONFLICT ON for pointers with DML in them (#4357)
- Fix cardinality in CommandDataDescription (#4347)
- Prevent access rule hidden ids from leaking when accessed directly (#4339)
- Better messages for required links hidden by policies (#4338)
- Fix access policies on DELETE of a UNION type (#4337)
- Strip out all views from DML subjects when computing what tables to use (#4336, #4333)
- Fix interaction between access policies and omitted fields in insert (#4332, #4219)
- Fix a tracer issue with reverse links and IS (#4331)
- Don't include union types in link triggers (#4329, #4320)

If you encounter this issue, after upgrading to a version with this patch, it can be fixed by doing a dump/restore or by adding a new link to the affected type.

- Require ON for constraints on objects (#4324, #4268)
- Fix interaction between DETACHED and aliases/globals (#4321, #4258)
- Disable access policy rewrite when compiling constraints (#4248, #4245)

- Expose `--admin-ui` as an environment variable and document it ([#4255](#))
- Prevent `HttpProtocol.close` from crashing on closed client connection ([#4238](#))
- Fix permitted JSON null in nested array cast ([#4221](#))
- Fix `range_unpack` boundary bug.

The `range_unpack` function was incorrectly excluding values close to boundary, especially when the boundary was not itself inclusive. ([#4282](#))

- UI: Allow selection of read-only properties in data editor ([edgedb/edgedb-ui/#65](#))
- UI: Hide subtype columns in data editor by default; add a toggle to show them. ([edgedb/edgedb-ui/#43](#))
- UI: Add “create example database” to the database selection screen. ([edgedb/edgedb-ui/#61](#))
- UI: Fix navigation from being reset on switching the UI panes. ([edgedb/edgedb-ui/#61](#))
- UI: Fix rendering of range types. ([edgedb/edgedb-ui/#61](#))
- UI: Fix the data editor UI to render types that have some properties or links masked by an access policy. ([edgedb/edgedb-ui/#61](#))
- UI: Implement login page for remote instances. ([edgedb/edgedb-ui/#40](#))

## 9.2.6 2.3

- Clarify error message when UI is not enabled ([#4256](#))
- Fix an issue with inherited computeds ([#4371](#))
- Fix bug in diamond pattern constraint inheritance ([#4379](#))
- When finding common parent for arrays, never use expr alias arrays ([#4080](#))
- Properly quote numeric names when in codegen ([#4344](#))
- Fix computed global scoping behavior ([#4388](#))
- Fix DDL performance issues on databases with lots of data ([#4401](#))
- Fix potentially missed constraints on DML ([#4410](#))
- Fix slicing with an empty set ([#4404](#))
- Fix slicing array of tuples ([#4391](#))
- Don’t apply access policies when compiling indexes ([#4420](#))
- Fix slicing of tuple arrays with null inputs ([#4421](#))
- Propagate database creation and deletion events to adjacent servers ([#4415](#))

## 9.2.7 2.4

- Fix database initialization on hosted environments like Heroku. ([#4432](#))
- Prevent spurious errors when using backlinks on types that have properties with the same name but different types ([#4443](#))
- Fix some spurious errors when removing a link from the schema. ([#4451](#))
- For query\_single, only check that the *last* query in a script is single. ([#4453](#))
- Catch when POPULATE MIGRATION generates incorrect DDL. This should prevent bugs where the schema can get into wedged states. ([#4484](#))
- workflows: Publish multiarch Docker images ([#4486](#))
- Make unused param insertion in the sql compiler more reliable ([#4497](#))
- Properly propagate creation and deletion of extensions
- Fix potential exclusive constraint violations when doing an UPDATE on a union ([#4507](#))
- Don't lose type from inheritance views when rebasing ([#4509](#))
- Make object type descriptor ids be derived from type name ([#4503](#))
- Check for invalid arrays arguments at the protocol level ([#4511](#))
- Fix SET REQUIRED on newly created properties with alias subtypes ([#4513](#))
- Make newly created link properties get added to the relevant alias types ([#4512](#))
- Fix handling of link properties named id ([#4514](#))
- Disallow queries using conflict machinery on a link property. This prevents certain potential exclusive constraint violations that were not handled correctly. ([#4515](#))
- Fix performing multiple deletions at once in the UI ([#4523](#))
- Fix casting empty sets to built in enum types ([#4532](#))
- Produce better error messages when using enum incorrectly ([#4527](#))
- Make '\b' produce the correct value in string and bytes literals ([#4535](#))

## 9.2.8 2.5

- Properly infer cardinality of empty array as ONE ([#4533](#))
- Fix several issues that manifest when using GROUP BY ([#4549](#), [#4439](#))
- Fix migration scripts when combined with access policies ([#4553](#))
- Fix failure when a ALTER . . . EXTENDING doesn't change the set of ancestors ([#4554](#))
- Fix UNLESS CONFLICT ON for a not-inserted property ([#4556](#))
- Fix access policies that use shapes internally ([#4555](#))
- Allow overloading \_\_type\_\_ with a computed in shapes ([#4557](#))

## 9.2.9 2.6

### 9.2.9.1 Nonrecursive access policies and future behaviors

Starting with EdgeDB 3.0, access policy restrictions will **not** be applied while evaluating other access policy expressions ([#4574](#)).

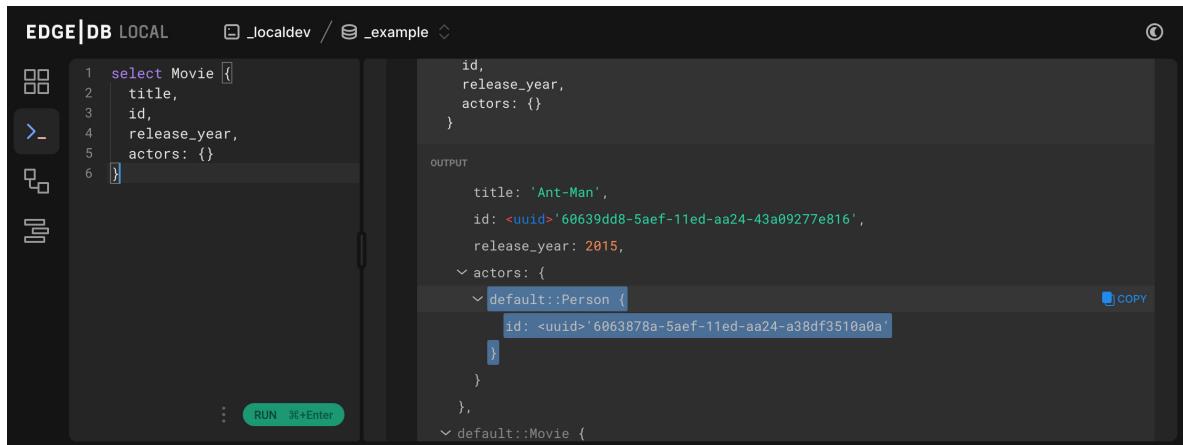
It is possible (and recommended) to enable this *future* behavior in EdgeDB 2.6 by adding the following to the schema: `using future nonrecursive_access_policies;`

For more details, see [the docs](#).

To enable opting in to this behavior, 2.6 adds a general mechanism to opt into *future* behavior changes ([#4574](#), [#4606](#)).

### 9.2.9.2 Other changes

- Fix passing zero dimensional array as arguments. This was a regression introduced in 2.4, and affected passing empty arrays from the the Rust bindings. ([#4511](#))
- Require that constraint expressions be immutable ([#4593](#))
- Only permit valid UUID-generation functions to be made the default value for `id` ([#4616](#))
- UI: New mechanism for copying data in REPL and in Data Editor. Hover over a data line and click the context “COPY” button.



- UI: “Disable Access Policies” and “Persist Query” options in REPL remember their state between page refreshes.
- UI: Basic autocomplete now works for `INSERT`, `UPDATE`, and `DELETE` queries.

## 9.2.10 2.7

- Improve error messages when compiling pointer default ([#4624](#))
- Fix using `WITH`-bound DML from an `UPDATE` in an `ELSE` clause ([#4641](#))
- Allow `WITH MODULE` in `ddl` in `CREATE MIGRATION` ([#4668](#))
- Fix some broken casts from object types to JSON ([#4663](#))
- Fix putting a statement as the body of an access policy ([#4667](#))
- Loosen the rules on when we produce a “would change the interpretation” error. It is not only produced when a link is being used, not a property. ([#4643](#))

- Fix certain errors involving default values in access policies ([#4679](#))
- Avoid ISE when pickling DynamicRangeVar ([#4681](#))
- Fix `max_ex_value` constraint. ([#4671](#))
- Fix SET GLOBAL capabilities to no longer break in the CLI. ([#4688](#))
- Fix links to `schema::ObjectType` breaking DROP TYPE. If you have a link in your schema to `schema::ObjectType` or one of its ancestors and you encounter internal server errors when trying to drop a type, it should be possible to repair your database by creating and then deleting a new link to `schema::ObjectType`. ([#4670](#))
- Don't insert unnecessary `assert_exists` calls on required links inside access policies bodies in some cases. ([#4695](#))

## 9.2.11 2.8

- Fix DML access policies that use shapes internally ([#4589](#))
- Give a proper error message creating a migration with a USING that has DML ([#4707](#))
- Don't incorrectly evaluate DML access policies when elements are also DML. This fixes some cases in which policies would pass incorrectly. ([#4745](#))
- Fix direct use of `__subject__` from insert access policies ([#4752](#))
- Produce an error message on casts to and literal references of enum types from indexes and constraints. Currently we generate an internal server error. A real fix unfortunately must wait for 3.0 for technical reasons. ([#4754](#))
- Only apply filter cardinality inference to unique sets ([#4763](#))
- Fix changing a link to non-computed and single at the same time ([#4764](#))
- Fix error message for function calls on derived types ([#4757](#))
- Fix deleting certain complex aliases ([#4777](#))
- Fix link properties on inherited backlinks ([#4788](#))
- Fix using array/string/bytes/json subscripting inside of indexes and constraints. ([#4760](#))
- Fix apparent startup hangs due to a lock fd leaking into postgres ([#4797](#))
- Fix some migrations with tricky constraint/computed interactions ([#4794](#))

## 9.2.12 2.9

- Fix broken DROPs of pointers in some multiple-inheritance situations ([#4809](#))
- Properly execute function calls in UPDATE once per object ([#4810](#))
- Fix accessing tuple elements on link properties ([#4811](#))
- Fix `assert_exists()` not firing on some tuple values ([#4812](#))
- Fix GROUP on the result of enumerate ([#4813](#))
- Support more env vars in args.py, standardize docs ([#4387](#))
- Fix computed properties that just copy id ([#4807](#))
- Fix backlinks on derived union types ([#4818](#))
- Fix references to the enclosing type in schema-defined computeds ([#4826](#))

- UI: Fix regression introduced in 2.8 when editing empty string fields in data explorer
- UI: Improvements to handling of union link targets in schema and data explorer views
- UI: Fix loading indicators on tabs

### 9.2.13 2.10

- Fix mismatch in session state after a ROLLBACK
- Fix ISE when doing set default on an abstract pointer ([#4843](#))
- Fix accesses to `__type__` from insert access policies ([#4865](#))
- Properly forbid aggregation in index expressions ([#4869](#))
- Fix grouping field when grouping by one key or nothing ([#4906](#))
- Fix two issues with mutation in free objects ([#4902](#))
- Only allow type names as the subject of an insert. (Previously dotted paths were allowed, with nonsensical behavior.) ([#4922](#))
- Fix array arguments in HTTP interface ([#4956](#))
- Support multi properties in UNLESS CONFLICT ON ([#4955](#))
- Fix polymorphic type tests on result of update ([#4954](#))
- Optimize trivial WITH -bound GROUP uses ([#4978](#))
- Fix a category of confusing scoping related bugs in access policies ([#4994](#))
- Get rid of the “unused alias definition” error. ([#4819](#))
- Support mutation in USING expressions when changing a link to required or to single during a migration ([#4873](#))
- Fix custom function calls on the HTTP interface ([#4998](#))
- Avoid infinite recursion in some do-nothing intersection cases ([#5007](#))
- Don’t mangle cast error messages when the cast value contains a type name ([#5008](#))
- Allow JWT token auth in binary protocol ([#4830](#))
- Use prepared statement cache in EdgeQL script execution ([#4931](#))
- Fix non-transactional commands like DROP DATABASE when using Postgres 14.7 ([#5026](#))
- Update packaged Postgres to 14.7
- Fix `set single` on required properties ([#5031](#))
- Fix a ISE when using assert\_exists and linkprops using query builder ([#4961](#))

## 9.2.14 2.11

- Fix adding a link property with a default value to an existing link (a regression in 2.10) ([#5061](#))

## 9.2.15 2.12

- Fix GROUP regression with some query-builder queries (a regression in 2.10) ([#5073](#))

## 9.2.16 2.13

- Implement a MIGRATION REWRITE system. This provides a mechanism for safely rewriting the migration history of a database while ensuring that the new history produces the same result as the old history. CLI tooling to take advantage of this feature is coming soon. ([#4585](#))
- Fix DigitalOcean support: allow its custom error in bootstrap ([#5139](#))
- Add a hint to the error message about link targets. ([#5131](#))
- Infer cardinality of `required multi` pointers as AT\_LEAST\_ONE ([#5180](#))
- Fix dump/restore of migrations with messages on them ([#5171](#))
- Fix interaction of link properties and `assert_exists` and similar ([#5182](#))
- Make `assert_single` and similar not lose track of values updated in an UPDATE in their argument. ([#5088](#))
- Add a test for `assert_exists+assert_single+UPDATE` ([#5242](#))
- Add support for new JWT layout ([#5197](#))
- Fix uses of volatile expressions in update write access policies ([#5256](#))
- Allow `globals` to be used in defaults ([#5268](#))
- Fix errmessage interpolation to not produce an internal server error on braces in a message. Allow `\{\}` to be used to escape braces. ([#5295](#))

## 9.2.17 2.14

### 9.2.17.1 Schema repair on upgrades

Previously, certain bug fixes and changes (such as the fix to cardinality inference of `required multi` pointers released in 2.13 ([#5180](#))), could cause schemas to enter an inconsistent state from which many migrations were not possible.

The cause of this problem is that an incorrectly inferred value (such as the cardinality of a computed property) may have been computed and stored in a previous version. When a newer version is used, there will be a mismatch between the correctly inferred value on the new version, and the incorrectly stored value in the database's schema.

The most straightforward way to fix such problems was to perform a dump and then a restore.

To fix this, we have introduced a schema repair mechanism that will run when upgrading a database to 2.14. This repair mechanism will fix any incorrectly inferred fields that are stored in the schema.

One particular hazard in this, however, is that the repair is not easily reversible if you need to downgrade to an earlier version. **We recommend performing a dump before upgrading to 2.14.**

These changes were made in ([#5337](#)); more discussion of the issue can be found in ([#5321](#)).

### 9.2.17.2 Other changes

- Correctly display constraint errors on `id` ([#5344](#))
- Fix adding certain computed links to a type with an alias ([#5329](#))

### 9.2.18 2.15

- In multi-server instances, properly reload schema after a restore ([#5463](#))
- Fix several bugs synchronizing configuration state
- Fix dropping a pointer's constraint and making it computed at the same time ([#5411](#))
- Don't claim that making a pointer computed is data-safe ([#5412](#))
- Prohibit NUL character in query source ([#5414](#))
- Fix migration that delete an link alias computed in a parent and child ([#5428](#))
- Fix GraphQL updates for multi links. ([#4260](#))
- Fix altering enum that is used in a tuple ([#5445](#))
- Fix changing cardinality of properties on types used in unions ([#5457](#))
- Enable GraphQL support for type unions.
- Fix making pointer non-computed and giving it an abstract base at the same time ([#5458](#))
- Make json casts of object arrays not include extra fields ([#5484](#))
- Make coalesce infer a union type ([#5472](#))

## 9.3 v3.0 (dev)

**edb-alt-title** EdgeDB v3 (dev)

**Warning:** The latest stable EdgeDB release is 2.x. EdgeDB 3.0 stable will be released soon.



EdgeDB 3.0 is currently in pre-release. We would like to thank our community for reporting issues and contributing fixes. You are awesome!

To play with the new features, install the CLI using [our installation guide](#) and initialize a new project.

```
$ edgedb project init --server-version=3.0-rc.1
```

---

**Note:** Good news, everyone! Upgrades across pre-release versions of 3.0 and then from pre-release 3.0 to the final 3.0 release will *not* require a dump and restore. You can try out 3.0 with the assurance that your initial upgrade to any 3.0 pre-release version will be the last dump and restore that is required of you.

---

### 9.3.1 Upgrading

#### Local instances

To upgrade a local project, first ensure that your CLI is up to date with `edgedb cli upgrade`. Then run an upgrade check to make sure your schema will migrate cleanly to 3.0.

```
$ edgedb migration upgrade-check
```

---

**Note:** EdgeDB 3.0 fixes a bug that will cause it to care about the ordering of your ancestors in multiple inheritance. This used to work before 3.0:

```
type A;
type B extending A;
type C extending A, B;
```

but as of 3.0, the order of ancestors must be changed to match the order of the bases:

```
type A;
type B extending A;
type C extending B, A;
```

This is a key instance where schemas may be incompatible with 3.0.

---

If the upgrade-check finds any problems, fix them in your schema and squash your migrations.

```
$ edgedb migration create --squash
```

Then run the following command inside the project directory.

```
$ edgedb project upgrade --to-testing
```

Alternatively, specify an instance name if you aren't using a project.

```
$ edgedb instance upgrade --to-testing -I my_instance
```

#### Hosted instances

To upgrade a remote (hosted) instance, we recommend the following dump-and-restore process.

1. Spin up an empty 3.0 instance. You can use one of our [deployment guides](#), but you will need to modify some of the commands to use our testing channel and the beta release.

Under Debian/Ubuntu, when adding the EdgeDB package repository, use this command instead:

```
$ echo deb [signed-by=/usr/local/share/keyrings/edgedb-keyring.gpg] \
  https://packages.edgedb.com/apt \
  $(grep "VERSION_CODENAME=" /etc/os-release | cut -d= -f2) testing \
  | sudo tee /etc/apt/sources.list.d/edgedb.list
```

Use this command for installation under Debian/Ubuntu:

```
$ sudo apt-get update && sudo apt-get install edgedb-3-rc1
```

Under CentOS/RHEL, use this installation command:

```
$ sudo yum install edgedb-3-rc1
```

In any required `systemctl` commands, replace `edgedb-server-2` with `edgedb-server-3`.

Under any Docker setups, supply the `3.0-rc.1` tag.

2. Take your application offline, then dump your v2.x database with the CLI

```
$ edgedb dump --dsn <old dsn> --all --format dir my_database.dump/
```

This will dump the schema and contents of your current database to a directory on your local disk called `my_database.dump`. The directory name isn't important.

3. Restore the empty v3.x instance from the dump

```
$ edgedb restore --all my_database.dump/ --dsn <new dsn>
```

Once the restore is complete, update your application to connect to the new instance.

This process will involve some downtime, specifically during steps 2 and 3.

### Pre-1.0 Instances

If you're still running pre-1.0 EdgeDB instances (e.g., 1.0-beta3) and want to upgrade to 3.0, we recommend you upgrade to version 2.x first, followed by another upgrade to 3.0, both using the same dump-and-restore process.

#### 9.3.1.1 Client libraries

Many of the client libraries have gained code generation capabilities since our 2.0 release. Look for new releases of all of our client libraries soon which will support all 3.0 features.

### 9.3.2 New features

#### 9.3.2.1 Simplified SDL syntax

As part of our commitment to delivering the best developer experience in databases, we've made our schema definition language (or SDL) easier to use. You're no longer required to use the `property` or `link` keywords for non-computed properties and links. Also, we've replaced arrows with colons for a cleaner look that's easier to type.

---

**Note:** If you prefer the arrow syntax of pre-3.0, feel free to keep using it. That syntax is still fully supported.

---

This change paves the way for a future syntax for declaring ad-hoc types in queries and functions. (Read more about it in the [free types RFC](#).)

That means that this type definition:

```
type User {
    required property email -> str;
    multi link friends -> User;
}
```

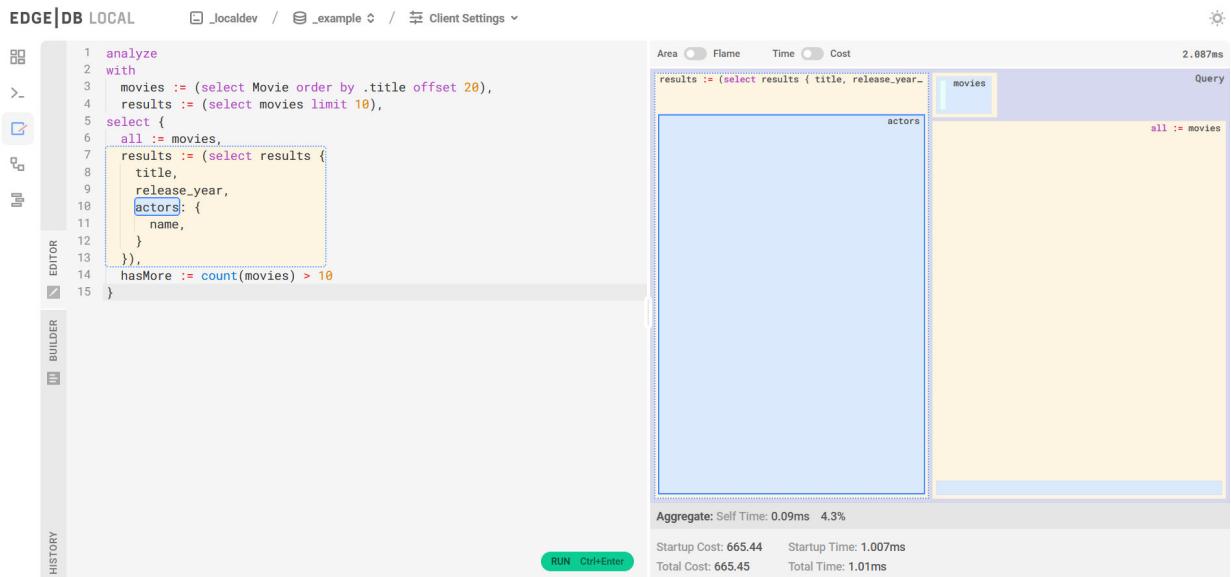
could be replaced with this equivalent one in EdgeDB 3+:

```
type User {
    required email: str;
    multi friends: User;
}
```

Selecting “v3” from the version dropdown in the sidebar will update SDL code in versioned sections of the documentation to the new syntax.

### 9.3.2.2 Query performance analysis

Among other improvements, the UI now includes a visual query analyzer to help you tweak performance on your EdgeQL queries. Just drop the `analyze` keyword in front of your query in the UI’s “Query Editor” tab to see the query analyzer in action.



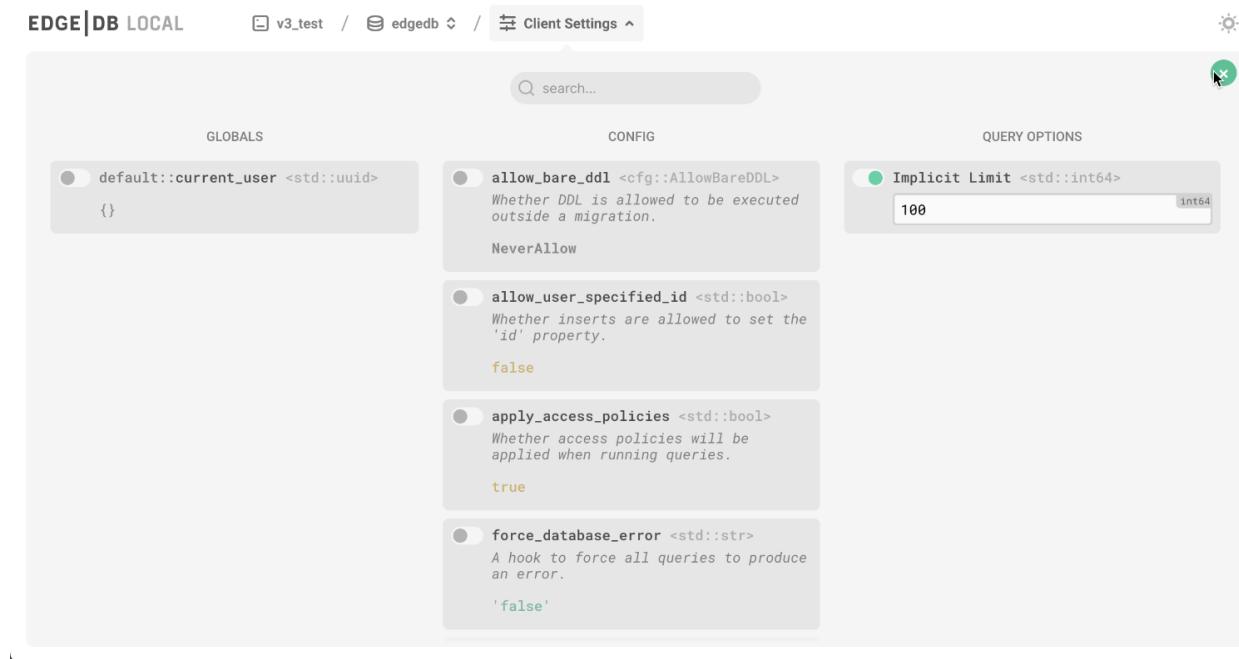
Query analysis is available in the CLI REPL by prepending your query with `analyze` or using the `\analyze` backslash command, and in the CLI directly using the `edgedb analyze <query>` command.

### 9.3.2.3 UI improvements

The EdgeDB UI got a lot of love in this release. In addition to the visual query planning shown above, you'll see a number of improvements.

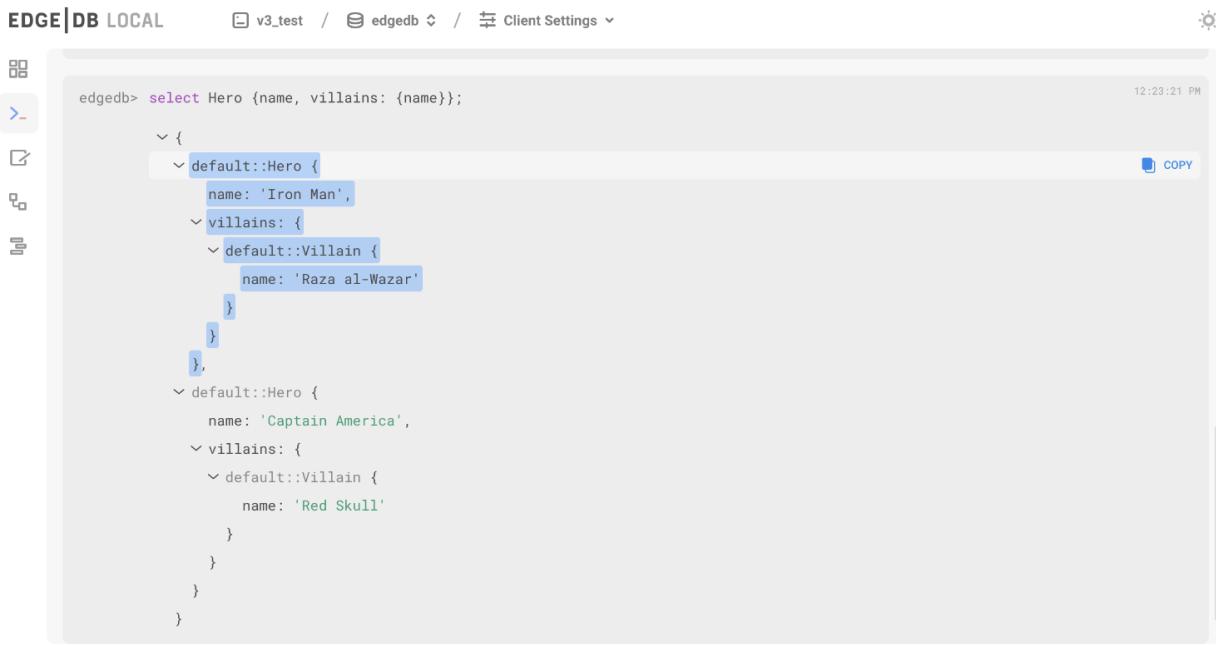
#### 9.3.2.3.1 New UI for setting globals and configuration

We've made it easier to set your globals and change configuration.



#### 9.3.2.3.2 New UI REPL

The UI's redesigned REPL makes it easy to drill into values and copy parts of your query results to the clipboard.



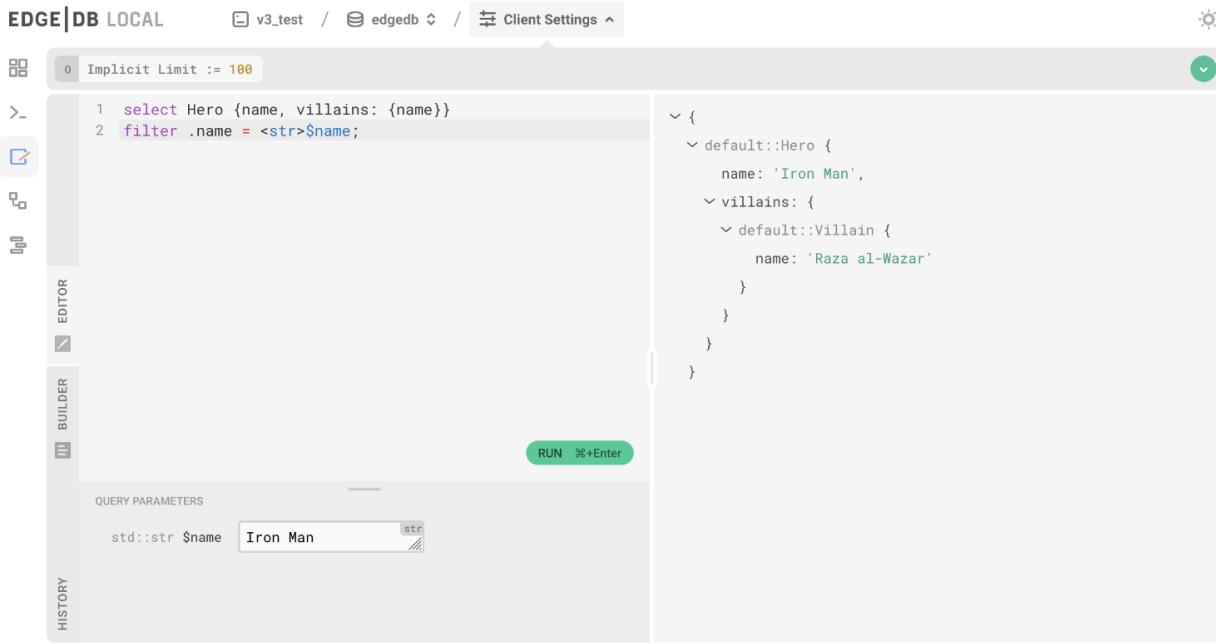
The screenshot shows the EdgeDB query editor interface. At the top, it says "EDGE|DB LOCAL" and "v3.test / edgedb / Client Settings". The main area displays a query result as a hierarchical tree. The root node is "Hero" with two children: "Iron Man" and "Captain America". Each hero has a "villains" child node. "Iron Man" has one villain: "Raza al-Wazar". "Captain America" has one villain: "Red Skull". A "COPY" button is visible in the top right corner of the results pane.

```

edgedb> select Hero {name, villains: {name}};
          ↓
        ↳ default::Hero {
            name: 'Iron Man',
            ↓ villains: {
                ↓ default::Villain {
                    name: 'Raza al-Wazar'
                }
            }
        },
        ↳ default::Hero {
            name: 'Captain America',
            ↓ villains: {
                ↓ default::Villain {
                    name: 'Red Skull'
                }
            }
        }
    
```

### 9.3.2.3.3 Query editor and visual builder

The query editor has a great new on-demand UI for setting parameters.



The screenshot shows the EdgeDB query editor with the "EDITOR" tab selected. It displays a query with a parameter "Implicit Limit := 100". The query itself is "select Hero {name, villains: {name}} filter .name = <str>\$name;". Below the query, the "BUILDER" tab is active, showing a "QUERY PARAMETERS" section with a single parameter "std::str \$name" set to "Iron Man". A "RUN ⌘+Enter" button is located at the bottom of the builder panel.

```

Implicit Limit := 100
1 select Hero {name, villains: {name}}
2 filter .name = <str>$name;

```

EDITOR

BUILDER

QUERY PARAMETERS

std::str \$name Iron Man

RUN ⌘+Enter

It also comes with a visual query builder which makes it easy to write queries, even when you're just learning EdgeQL.



### 9.3.2.4 edgedb watch and a new development workflow

The new `edgedb watch` CLI command starts a long-running process that watches for changes in schema files in your project’s `dbschema` directory and applies those changes to your database in real time. This command opens up an entirely new workflow for prototyping schema that will result in less migration clutter in your repositories.

#### 9.3.2.4.1 1. Start the watch command

```
$ edgedb watch
Initialized. Monitoring "/projects/my-edgedb-project".
```

#### 9.3.2.4.2 2. Write an initial schema

Just start writing your schema in your `default.esdl` file in your project’s `dbschema` directory. Once you save your initial schema, assuming it is valid, the `watch` command will pick it up and apply it to your database.

#### 9.3.2.4.3 3. Edit your schema files

As your application evolves, directly edit your schema files to reflect your desired data model. When you save your changes, `watch` will immediately begin applying your new schema to the database.

Once you have the schema the way you want it, you’re ready to lock it in by generating a migration.

#### 9.3.2.4.4 4. Generate a migration

To generate a migration that reflects all your changes, run `edgedb migration create`.

```
$ edgedb migration create
```

This “locks in” the changes you prototyped using the `watch` command. Now, these are ready to commit and push to your remote to share with your team.

### 9.3.2.5 Triggers

Our new triggers feature is one of the most anticipated 3.0 features! Triggers allow you to define an expression to be executed whenever a given query type is run on an object type. The original query will *trigger* your pre-defined expression to run in a transaction along with the original query. These can be defined in your schema.

```
type Person {
    required name: str;

    trigger log_insert after insert for each do (
        insert Log {
            action := 'insert',
            target_name := __new__.name
        }
    );
}
```

The trigger above inserts a Log object any time a Person object is inserted.

You can read more about our triggers implementation in the [triggers RFC](#).

### 9.3.2.6 Mutation rewrites

The mutation rewrites feature is the sibling, or at least the first cousin, of triggers. Both are automatically invoked when a write operation occurs on the type they're on, but triggers are not able to make changes to the object that invoked them. Mutation rewrites are built to do just that!

```
type Post {
    required title: str;
    required body: str;
    modified: datetime {
        rewrite insert, update using (datetime_of_statement())
    }
}
```

This shows one reason mutation rewrites is one of our most wanted features: modified timestamps! When a user inserts or updates a Post, the rewrite will set the value of the modified property to that value of `datetime_of_statement()`. There are tons of other uses too. Give them a try!

Learn about our mutation rewrites implementation in [the mutation rewrites RFC](#).

### 9.3.2.7 Splits

This is one of the most fun features in 3.0, both to say `_and_` to use! With splats, you can easily select all properties in your queries without typing all of them out.

Before splats, you would have needed this query to select Movie objects along with all their properties:

```
select Movie {id, release_year, title, region, director, studio};
```

Now, you can simplify down to this query instead using a splat:

```
select Movie {*};
```

If you wanted to select the movie and its characters before splats, you would have needed this:

```
select Movie {  
    id,  
    release_year,  
    title,  
    region,  
    director: {id, name, birth_year},  
    actors: {id, name, birth_year},  
    characters: { id, name }  
};
```

Now, you can get it done with just a double-splat to select all the object's properties and the properties of any linked objects nested a single layer within it.

```
db> select Movie {**};
```

It's a super-handy way to quickly explore your data.

Read more about splats in [our splats RFC](#).

### 9.3.2.8 SQL support

EdgeDB supports running read-only SQL queries via the Postgres protocol to enable connecting EdgeDB to existing BI and analytics solutions. Any Postgres-compatible client can connect to your EdgeDB database by using the same port that is used for the EdgeDB protocol and the same database name, username, and password you already use for your database.

```
$ psql -h localhost -p 10701 -U edgedb -d edgedb
```

Our SQL support has been tested against a number of SQL tools:

- [pg\\_dump](#)
- [Metabase](#)
- [Cluvio](#)
- [Tableau](#)
- [DataGrip](#)
- [Airbyte](#)
- [Fivetran](#)
- [Hevo](#)
- [Stitch](#)
- [dbt](#)

### 9.3.2.9 Nested modules

You can now put a module inside another module to let you organize your schema in any way that makes sense to you.

```
module momma_module {
    module baby_module {
        # <schema-declarations>
    }
}
```

In EdgeQL, you can reference entities inside nested modules like this: `momma_module::baby_module::<entity-name>`

Aside from giving you additional flexibility, it will also allow us to expand our list of standard modules in a backwards-compatible way.

### 9.3.2.10 intersect and except operators

Slice and dice your sets in new ways with the `intersect` and `except` operators. Use `intersect` to find common members between sets.

```
db> select {1, 2, 3, 4, 5} intersect {3, 4, 5, 6, 7};
{3, 5, 4}
```

Use `except` to find members of the first set that are not in the second.

```
db> select {1, 2, 3, 4, 5} except {3, 4, 5, 6, 7};
{1, 2}
```

These work with sets of anything, including sets of objects.

```
db> with big_cities := (select City filter .population > 1000000),
...     s_cities := (select City filter .name like 'S%')
... select (big_cities intersect s_cities) {name};
{default::City {name: 'San Antonio'}, default::City {name: 'San Diego'}}
db> with big_cities := (select City filter .population > 1000000),
...     s_cities := (select City filter .name like 'S%')
... select (big_cities except s_cities) {name};
{
    default::City {name: 'New York'},
    default::City {name: 'Los Angeles'},
    default::City {name: 'Chicago'},
    default::City {name: 'Houston'},
    default::City {name: 'Phoenix'},
    default::City {name: 'Philadelphia'},
    default::City {name: 'Dallas'}
}
```

### 9.3.2.11 assert function

The new `assert` function lets you do handy things like create powerful constraints when paired with triggers:

```
type Person {
    required name: str;
    multi friends: User;
    multi enemies: User;

    trigger prohibit_frenemies after insert, update for each do (
        assert(
            not exists (__new__.friends intersect __new__.enemies),
            message := "Invalid frenemies",
        )
    )
}
```

```
db> insert Person {name := 'Quincey Morris'};
{default::Person {id: e4a55480-d2de-11ed-93bd-9f4224fc73af}}
db> insert Person {name := 'Dracula'};
{default::Person {id: e7f2cff0-d2de-11ed-93bd-279780478afb}}
db> update User
... filter .name = 'Quincey Morris'
... set {
...     enemies := (select Person filter .name = 'Dracula')
... };
{default::Person {id: e4a55480-d2de-11ed-93bd-9f4224fc73af}}
db> update User
... filter .name = 'Quincey Morris'
... set {
...     friends := (select Person filter .name = 'Dracula')
... };
edgedb error: EdgeDBError: Invalid frenemies
```

You can use it in other contexts too — any time you want to throw an error when things don't go as planned.

## 9.3.3 Additional changes

### 9.3.3.1 EdgeQL

- Support custom user-defined error messages for access policies (#4529)

```
type User {
    required property email -> str { constraint exclusive; };
    required property is_admin -> bool { default := false };
    access policy admin_only
        allow all
        using (global current_user.is_admin ?? false) {
            errmessage := 'Only admins may query Users'
        };
}
```

- Support casting a UUID to a type (#4469). This is a handy way to select an object, assuming the type you cast into has an object with the UUID being cast.

```
db> select <Hero><uuid>'01d9cc22-b776-11ed-8bef-73f84c7e91e7';
{default::Hero {id: 01d9cc22-b776-11ed-8bef-73f84c7e91e7}}
```

- Add the `json_object_pack()` function to construct JSON from an array of key/value tuples. (#4474)

```
db> select json_object_pack({{"hello", <json>"world")});
{Json("{\"hello\": \"world\"})}
```

- Support tuples as query arguments (#4489)

```
select <tuple<str, bool>>$var;
select <optional tuple<str, bool>>$var;
select <tuple<name: str, flag: bool>>$var;
select <optional tuple<name: str, flag: bool>>$var;
select <array<tuple<int64, str>>>$var;
select <optional array<tuple<int64, str>>>$var;
```

- Add the syntax for abstract indexes (#4691)

Exposes some Postgres indexes that you can use in your schemas. These are exposed through the `pg` module.

- `pg::hash`- Index based on a 32-bit hash derived from the indexed value
- `pg::btree`- B-tree index can be used to retrieve data in sorted order
- `pg::gin`- GIN is an “inverted index” appropriate for data values that contain multiple elements, such as arrays and JSON
- `pg::gist`- GIST index can be used to optimize searches involving ranges
- `pg::spgist`- SP-GIST index can be used to optimize searches involving ranges and strings
- `pg::brin`- BRIN (Block Range INdex) index works with summaries about the values stored in consecutive physical block ranges in the database

Learn more about the index types we expose [in the Postgres documentation](#).

You can use them like this:

```
type User {
    required property name -> str;
    index pg::spgist on (.name);
};
```

- Implement migration rewrites (#4585)
- Implement schema reset (#4714)
- Support link properties on computed backlinks (#5227)

### 9.3.3.2 CLI

- Add the `edgedb migration upgrade-check` command

Checks your schema against the new EdgeDB version. You can add `--to-version <version>`, `--to-testing`, `--to-nightly`, or `--to-channel <channel>` to check against a specific version.

- Add the `--squash` option to the `edgedb migration create` command

This squashes all your migrations into a single migration.

- Change the backslash command `\d object <name>` to `\d <name>`

- Add the `edgedb migration edit` command ([docs](#); released in 2.1)

- Add the `--get` option to the `edgedb info` command (released in 2.1)

Adding the `--get` option followed by a name of one of the info values — `config-dir`, `cache-dir`, `data-dir`, or `service-dir` — returns only the requested path. This makes scripting with the `edgedb info` command more convenient.

### 9.3.3.3 Bug fixes

- Fix crash on cycle between defaults in insert ([#5355](#))
- Improvements to top-level server error reporting ([#5349](#))
- Forbid ranges of user-defined scalars ([#5345](#))
- Forbid DML in non-scalar function args ([#5310](#))
- Don't let "owned" affect how we calculate backlinks ([#5306](#))
- Require inheritance order to be consistent with the specified base order ([#5276](#))
- Support using non-strict functions in simple expressions ([#5271](#))
- Don't duplicate the computation of single links with link properties ([#5264](#))
- Properly rebase computed links when changing their definition ([#5222](#))
- Fix 3-way unions of certain types with policies ([#5205](#))
- Fix simultaneous deletion of objects related by multi links ([#5201](#))
- Respect `enforce_access_policies := false` inside functions ([#5199](#))
- Fix inferred link/property kind when extending abstract link ([#5196](#))
- Forbid `on target delete deferred restrict` on required links. ([#5189](#))
- Make `uuidgen` properly set versions in `uuid4/uuid5` ([#5188](#))
- Disallow variadic arguments with optional types in user code. ([#5110](#))
- Support casting between scalars with a common concrete base ([#5108](#))
- Fix GROUP regression with some query-builder queries ([#5071](#))
- Fix a ISE when using `assert_exists` and `linkprops` using query builder ([#5036](#))
- Fix bug that dropping non-existing db leaves with unaccessible state ([#5032](#))
- Fix non-transactional errors in Postgres 14.7 ([#5028](#))
- Properly cast to containers of enums when loading from the schema ([#4988](#))
- Implement manual error override configuration ([#4974](#))

- Fix protocol state confusion after rollback ([#4970](#)), ([#4953](#))

#### 9.3.3.4 Deprecations

The support of version pre-1.0 binary protocol is deprecated in EdgeDB 3.0, and will be completely dropped in EdgeDB 4.0. If you're still using a deprecated version of the binary protocol or any client libraries that *only* support the pre-1.0 binary protocol as listed below, please consider upgrading to a newer version.

- edgedb-js / edgedb-deno v0.20 or lower
- edgedb-python v0.23 or lower
- edgedb-go v0.10 or lower
- edgedb-tokio (Rust) v0.2 or lower
- EdgeDB.NET v0.2 or lower
- edgedb-elixir v0.3 or lower

#### 9.3.4 New release schedule

Unfortunately, the 3.0 release will not include full-text search. We have many requirements for this new API (see [the FTS RFC](#) for details), and, while we've made significant progress, we have unfortunately run out of time to be 100% sure that it is ready for prime time.

We don't want this delay to hold back the release of EdgeDB 3.0, which includes many other exciting features that are ready for you to start using right now. That's why we've decided to delay only the FTS feature rather than delaying the entire 3.0 release.

That said we're working hard to get FTS ready as soon as possible. After the release of 3.0, we'll be moving to a much more frequent release cycle so that features like FTS can be in your hands as soon as they're ready.

Going forward, expect EdgeDB releases every four months. These releases will naturally incorporate fewer features than our past releases, but we think the more predictable cadence will be worth it. Every third release starting with 3.0 will be a long-term support (LTS) release. These releases will continue to receive support for a year and a half after their initial release.

#### 9.3.5 3.0 RC 1

##### 9.3.5.1 Changes to new 3.0 features

- Fix indirect references to properties in triggers ([#5450](#))
- Fix rewrites of aliases types ([#5461](#))
- Do nicer translation of relation names in ANALYZE ([#5467](#))
- Fix schema ordering bug when INSERT appears in triggers ([#5489](#))
- Produce correct error source locations in the SQL interface ([#5462](#))
- Fix deleting a pointer with rewrites in a migration ([#5503](#))

### 9.3.5.2 Other changes and fixes

- Support disabling dynamic configuration of system config (#5425)
- In multi-server instances, properly reload schema after a restore (#5463)
- Fix several bugs synchronizing configuration state
- Fix dropping a pointer's constraint and making it computed at the same time (#5411)
- Don't claim that making a pointer computed is data-safe (#5412)
- Prohibit NUL character in query source (#5414)
- Fix migration that delete an link alias computed in a parent and child (#5428)
- Fix GraphQL updates for multi links. (#4260)
- Fix altering enum that is used in a tuple (#5445)
- Fix changing cardinality of properties on types used in unions (#5457)
- Enable GraphQL support for type unions.
- Fix making pointer non-computed and giving it an abstract base at the same time (#5458)
- Make json casts of object arrays not include extra fields (#5484)
- Make coalesce infer a union type (#5472)
- Fix graphql queries made against a freshly started server (#5456)
- Fix version for project init (#5460)
- Produce a proper error for too many constraint args (#5454)
- Prevent using leading dot notation in inserts (#5142)
- Fix operations on single link with only computable link properties (#5499)
- Don't ISE on free shape in insert (#5438)
- Work around postgres server crashes on Digital Ocean during edgedb setup (#5505)
- Always set cardinality of derived \_\_tname\_\_ and \_\_tid\_\_ pointers (#5508)
- Make failovers more resilient (#5511)

## 9.4 Deprecation Policy

- We continue to support one previous version of EdgeDB with critical bug fixes.
- Client bindings will support the current and the previous major version.
- CLI supports all versions from version 1.