

## Functions in PGPLSQL

```
functions in psql
create or replace function funcName()
returns s returntype as $$
begin
raise notice 'Hello Birb!';
end;
$#
language langName
The two $ are always necessary. Also note the
returns with an s and the language at the end.
Which MUST BE PROCEDURAL LANGUAGE,
so c++ doesn't work here.
```

Parameters are handled like in any language  
`func(x bigint, y bigint)`  
you can also define multiple return types  
`func(variadic a numeric[])`  
or a generic return  
`func(param anyelement)`

Variable Declaration:

```
returns void as $$
DECLARE
x bigint; y bigint;
BEGIN ....
```

Variable manipulation: `x := 6 + 4`

```
if: IF n = 0 THEN RETURN 1;
(optional)ELSE RETURN 2; END IF;
ELSIF also possible (note elseif not elseif)
```

```
case x when 1, 2 then msg := 'one or two'; end case;
essentially this checks if x is 1 OR 2
```

case when x between 0 and 10 then ....

similar but with a range, both can be

simulated by if else.

Exceptions: BEGIN z := x / y;

EXCEPTION WHEN division-by-zero

THEN z:= 0; (or error rather) END;

if you want to catch all: WHEN others THEN

often used after exception: RAISE; (show error)

For Loop: For var IN query LOOP

```
statements END LOOP;
for r in SELECT * FROM ang LOOP
RETURN NEXT r; END LOOP; RETURN; END;
note that the return next doesn't return
```

you store it in buffer

and return it at the end of the function.

for infinite loops: FOR i IN 1.. max LOOP;

update and insert: INSERT INTO ANG VALUES(...);

UPDATE ang set salary = salary + 500

where name = 'dashie';

interestingly, after the where name = 'dashie'

you can use if not found then (handle error)

this allows for easier error handling.

queries: execute 'SELECT \* from ang'

|| into result; return result; END;

comments are done by either -- or /\* \*/ for multiline

anonymous function: you can omit

the name and just write do \$\$ ...

cursor: declare curs CURSOR FOR query;

BEGIN OPEN curs; LOOP do something CLOSE

curs; END;

Cursors are essentially just iterables.

curors can also be unbound curs1 refcursor

or they can be parameterized curs3 cursor(arg)

## PL/pgSQL: Datentypen

- Boolean:
- Zahlen: int, integer, number
- String, Datum, etc.
- Arrays: alle Datentypen gefolgt von "[]", z.B. int[]
- Weitere: JSON, etc.
- ... ergänzt mit zusätzlichen Datentypen:
  - var5 angesterter.IdType; - abgeleiterter col-Type
  - var6 angewählter%rowtype; - der abgeleitete von Tabelle
  - var7 record; - generischer Record This is a simple entry
  - var8 anyelement; - generischer Typ gemäß Fn.-Argument, vgl. nachfolgend
  - curs1 refcursor; curs2 cursor ...; - vgl. nachfolgend

arrays: SELECT '1,2,3':int[]
or SELECT ARRAY[1,2,3]
var int[] only in variable declaration.
!!arrays start with 1 in psql !!

return types: all of the above AND void.

SETOF type (array of a type), TABLE, Trigger

## Arrays: Accessoren

```
create table tictactoe as
(select * as id,
array[ [ [1,1], [1,2], [1,3] ],
[ [2,1], [2,2], [2,3] ],
[ [3,1], [3,2], [3,3] ] ] as board);
```

Index Query: intuitiv wie eine Koordinate ("1-basiert": Start mit 1 nicht 0):

```
select board[1][1] from tictactoe;
```

```
-- [1][1]
```

Slice Query: "Untergrenze-Obergrenze" für jede Dimension:

```
select board[2:3][1:1] from tictactoe;
```

```
-- [ [2,1], [2,2] ]
```

Max-Bound-Abkürzung "[2]" vermeiden (Verwechslungsgefahr), besser "[1:2]"

```
select board[1:2][1] from tictactoe; -- [1][1]
```

```
-- [ [1,1], [1,2], [1,3] ]
```

Suche nach einem Element aus dem tictactoe where '2' k2 = any (board);

```
-- [1][1], [2][1], [3][1]
```

Arrays: Operatoren

- Is equal: @=
- SELECT ARRAY[1,2,3] = ARRAY[1,2,3];
- SELECT ARRAY[1,2,3] <= ARRAY[1,2,3];
- SELECT ARRAY[1,2,3] >= ARRAY[1,2,3];
- SELECT ARRAY[1,2,3] < ARRAY[1,2,4,5,6];
- SELECT ARRAY[1,2,3] > ARRAY[1,2,4,5,6];

Contains: @<

```
- SELECT ARRAY[1,2,6] @< ARRAY[2,7];
```

is contained by: @<

```
- SELECT ARRAY[2,7] @< ARRAY[1,7,4,2,6];
```

Overlaps: @&

```
- SELECT ARRAY[1,4,3] @& ARRAY[2,1]
```

hstore / map:

Create and Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);
```

```
INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'):hstore, NULL);
```

```
SELECT * FROM test;
```

```
| id | col2 | col3 |
1 | "a"=>"123", "b"=>"foo", "c"=>"bar" | null |
(1 row)
```

Queries:

- List all keys
- SELECT akeys(mkvfield) FROM ...

- Get all key-value pairs
- SELECT each(mkvfield) FROM ...

- Get key value (as text)
- SELECT mvkfield->name' FROM ...

- Test if left hstore is contained in right hstore:

- WHERE mvkfield @> 'tourism'=>'zoo'; -- or hstore('tourism','zoo')

Operatoren:

- => get value for key: SELECT a=>x, b=>y:hstore->'a'
- @>, etc. ... ähnlich wie Array-Operator

hstore supports GIST/GIN indexing

	FUNCTION	PROCEDURE
Use in an expression	✓	✗
Return a value	✓	✗
Return values as OUT parameters	✓ (PG Spezialität) ✓ (PG 14)	
Return a single result set	✓ (as table fn.) ✓	
Return multiple result sets	✗	✓
Contain transactions	✗	✓
Make it run using ...	EXECUTE	CALL

also note that type%rowtype is used like this:  
`r ang%rowtype -> for r in select * from ang;`  
important to know, you can always use these functions to manipulate queries, for example  
`select upper(name) from ang;`  
depending on the function you can also  
`select generatetable(1,10)`

- Es gibt zusätzlich IN, OUT, INOUT
  - IN: call by value; Variablen oder Ausdrücke als Argument
  - OUT: call by reference; nur Variablen als Argument
  - INOUT: beides

cast: cast(input as type);

cast(record.id as text);

stored procedures are nothing but a chaining of functions:

Schrift 1 in PL/pgSQL: siehe Beispiel 3 (SP-Funktion mit in/out-Parameter)

Schrift 2: Deklaration in Java/JPA (aka Registrierung in JPA):

```
@NamedDBProcedureQuery
name = "MySum"
procedureName = "mysum", -- Name der SP-Fn. (DB-Objekt)
parameters = [
    @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "x"),
    @StoredProcedureParameter(mode = ParameterMode.IN, type = Double.class, name = "y"),
    @StoredProcedureParameter(mode = ParameterMode.OUT, type = Double.class, name = "sum")
]
```

Schrift 3: Call it!

```
StoredProcedureQuery query = this.em.createNamedStoredProcedureQuery("MySum");
query.setParameter("x", 1.23d);
query.setParameter("y", 4.56d);
query.execute();
Double sum = (Double) query.getOutputParameterValue("sum");
```

so good to know things: plain SQL is more efficient.

write variables lower case for sql

use cast over typename -> not select date '2022-06-07'

## Triggers

- sind DB-Objekte und immer einer Tabelle zugeordnet
- werden in Stored Procedures programmiert
- haben keine Parameter
- können nicht direkt aufgerufen werden
- werden vom DBMS beim Eintreten eines Events aufgerufen
- haben bei der Ausführung die Rechte ihres Owners

Trigger can pass parameters to function

FOR EACH [statement | row]

Events: INSERT, UPDATE, DELETE, TRUNCATE, INSTEAD OF

Function executes BEFORE or AFTER changes

statement is once, row means once per row, aka for the entire table.

\* Before triggers can change contents of new row

\* After triggers can only respond to what has happened e.g. Foreign Keys

\* Return values of AFTER triggers will be ignored

\* Triggers execute in alphabetical order

\* DDL für Triggers

```
CREATE | DROP | ALTER TRIGGER
```

```
mytrigger ...
ON mytable ...
```

Syntax-Beispiel CREATE TRIGGER:

```
CREATE TRIGGER mytrigger
```

```
AFTER INSERT OR UPDATE
```

```
ON mytable
```

```
FOR EACH ROW
```

```
EXECUTE PROCEDURE mytriggerfn();
```

The INSTEAD Trigger can be used to avoid crashes:

-> INSTEAD OF UPDATE

-> ON UPDATE DO INSTEAD

for example, trying to update a read-only view -> INSTEAD OF UPDATE

CREATE OR REPLACE FUNCTION dt\_trigger\_func()

RETURNS TRIGGER AS \$\$
BEGIN

IF (\$G\_OP = 'INSERT') THEN

NEW.creation\_date := now();

ELSIF (\$G\_OP = 'UPDATE') THEN

NEW.modification\_date := now();

END IF;

RETURN NEW;

BEFORE trigger

Falls RETURN null -->

\$\$\_ LANGUAGE plpgsql;

END

\$\$ LANGUAGE plpgsql;

wird Operation abgebrochen

Syntax-Beispiel Trigger-Fn. passend zu Trigger:

```
CREATE FUNCTION mytriggerfn()
```

```
RETURNS TRIGGER --
```

```
AS $$

```

```
</body>
$$_ language plpgsql;

```

- Eine Trigger-Fn. hat keine Parameter

- Diese werden über Trigger-Fn.-Variablen übergeben, u.a.:
  - TG\_NARGS Anzahl Parameter
  - TG\_ARGV[] Array von Parametern als TEXT

Return types: RETURN NEW -> returns new table/row

RETURN OLD -> returns the old table/row

(but could change other rows!)

RETURN NULL -> cancel operation.

running order: before statement, before row, after row, after statement -> and of course alphabetically.

inside the Trigger functions you can use the variables that don't matter aka can be ANY -> user or the entered user from the trigger -> NEW.user or explicitly the old one -> OLD.user and last user defined stuff like -> SELECT 'I'  
which just places an I as the variable or something like now() for timestamps.

Triggers make the database slower and harder to maintain.

some databases therefore let you disable them if you want.

On a table basis.

Also watch out for cascading effects of triggers they might cause something else to be deleted.

Stored Procedures are really helpful for security

They have all the privileges, but only

allow the user to do what the creator has predefined.

## User defined types in SQL

CREATE DOMAIN ...	• erstellt einen benutzerdefinierten Datentyp	• erstellt einen zusammengesetzten Datentyp
	• für Datenschema und Stored Procedures	• für ENUM Type verwendet
	• mit Constraints wie NOT NULL, CHECK	• Keine Constraints-Angebote
		create domain contact_name as varchar(255) not null check (value ~ '^[a-zA-Z0-9_]*\$');
		create type eastegg as { outer: text; inner: text; };
		create type traffic_light_t as enum('red', 'yellow', 'green');

One is a completely new type, the other is just a type that is made up of already known types.  
Essentially the right one is a class or struct.  
The left is a completely new thing, that is not yet implemented.

## Optimization and indexing

The basics of indexing is that is saves time on queries, but it uses more space, and needs to be redone on update/insert.

Data stored in Pages / Heap (Collection of Pages)

Indexing on either can be possible.

ex: Page index -> Primary key index

ex: Heap Index -> index on tables

PSQL does table clustered indexing instead of integrated indexing -> key value (both indexed)

B-Tree: the default, can index multiple entries (only btree!)

CREATE UNIQUE INDEX name ON table (column [1,2,...])

Hash Index: just like hashmap in programming

good for single or small multiple queries

bad for entire tables etc -> collision

hashing might take a long time with a lot of data.

B-tree almost always better!

GIST: balanced/tree-like , Range/neighbor/fulltext search

used for geometric datatypes. SP-Gist for unbalanced trees.

GIN: General Inverted Index "list of words

that point to documents" wtf? Good for duplicates.

Good for hstore,Json,Arrays

Vergleich zu GIST:

- Ca. 3x schneller Zugriff
- Ca. 2 - 3x mehr Diskplatz
- Ca. 2 - 3x länger bis Index erstellt ist
- Ca. 10x langsamer bei Update

Bitmap indexing: Bitmap -> 0 1 stores Booleans/Enums

very fast read / slow update

in postgres only implicit use Brin instead.

BRIN: Block Range Index, stores min/max values as blocks

good for range search, sorted data, small disk usage

data is naturally sorted, address next to postal code.

Bloom Index -> equality search,

Trigram Index -> Full text search

RUM , non-default-GIN jsonb-path-ops

creating index:

CREATE INDEX <indexname> ON <table(attribute)>;

and: DROP INDEX <indexname>;

default index order for psql is btree,ASC,NULL first

Index-Variationen

- Zusammengesetzter Index

- Bsp.: CREATE INDEX idx\_addr ON addr(phone,name); kann für Queries auf phone und "phone AND name" genutzt werden, sowie für bestimmte Q. auf name; jedoch nicht für Suffix-Q. "... LIKE '%name'"

- Index mit INCLUDE

- Bsp.: CREATE INDEX idx\_addr ON addr(phone) INCLUDE (name);

Index-Variationen ff.

- Partieller Index

- Bsp.: CREATE INDEX idx\_status ON addr(status) WHERE status='active';

- Index mit Funktionen / Ausdrücken ("Funktionaler")

- Bsp.: CREATE INDEX idx\_addr ON addr(lower(name));

- Nicht nur PostgreSQL

PG planner join strategies: Nested Loop, Merge, Hash

Nested Loop: for r in right row r == for l in left row....

good for small tables, easy to setup

Merge: Merge rows after the other

higher starting cost, good for bigger tables

Hash: Hash the row then compare to other row

equality check only , high starting cost, low execution cost

PG planner scans: Full,Index,Index Only, Bitmap

- full scans the entire table

- index scans index and more (if necessary)

- index only scans index

- Bitmap scans the bitmap generated by an index.

**fragmentation:** this is the splitting of schemas into multiple Nodes -> table 1 in node1 table 2 in node2.

In PSQL the horizontal fragmentation happens in 3 ways:  
 > 1 or more attributes for "partitioning key"  
 > "list" explicit designation  
 > hash function (ex: Modulo)

In Graph stores this would be called "sharding"  
 an example for this is the MongoDB horizontal partitioning and allocation within a single node

**replication:** this is the duplication of data in schemas this means table 1 might be on node1 and 2.

**vertical -> splitting of columns**

row1 in node1, row2 in node2

**horizontal -> splitting of rows**

part of column in node1 part of column in node2

**unidirectional:** Single-Master

**bidirectional:** Multi-Master

**synchronous OR asynchronous**

**allocation:** this is the distribution of work to the nodes node1 might handle query or part of query1 while node2 handles something else.

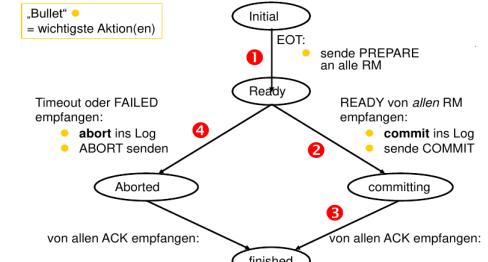
as already stated above, the user doesn't see anything about fragmentation or similar, the user simply interacts with the schema and executes transactions.

these transactions are always local

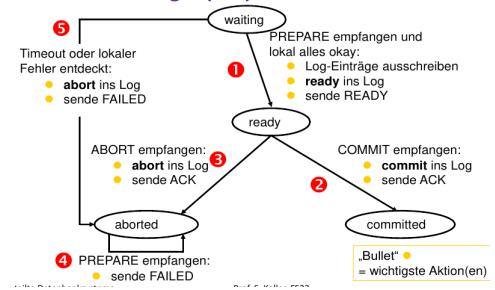
This means that the user will talk to **Transaction Manager** that will handle the transaction and call the necessary functions on the **Resource Managers** (inside nodes).

**Two Phase Commit Protocol**

### Transaction Manager TM



### Resource Manager (RM)



**case TM failed/restart:**

> if the TM crashes before the commit message -> abort  
 > if the TM crashes after RM respond ready -> block RM this is one of the main problems btw...

**case RM failed/restart:**

> if no entry in log, RM aborts  
 > if READY-Entry available -> RM asks TM what to do.  
 > if COMMIT-Entry available -> RM redo's transaction

**case message dropped:**

> if the prepare statement gets lost,  
 or the RM doesn't respond, then the TM simply aborts the transaction for all.

> if RM doesn't get a response in READY state

then the RM will remind the TM until it gets one.

There are different 2PC protocols, PSQL, MYSQL, X/Open, Java Transaction API, Oracle Transaction Manager, Microtroll + often used / proven system + guarantees ACID

- slow - blocks transactions often

!only use this when the complexity calls for it!!

**Query handling:**

there are 3 ways of handling a queries distributed databases:

> Ship Whole -> run query locally and remove duplicates  
 > Push Down -> split query when needed  
 > Fetch-as-needed -> Send join attributes to the correct node

### NoSQL: Key/Value Stores

Why even NoSQL? -> fast, lots of data, web-based, scalable RDBMS -> relational has the problem of complexity

it doesn't integrate seamlessly into programming languages

it therefore doesn't make sense to use this as

something like a webstore. -> !!! OR MAPPING !!!

**attributes of NoSQL:**

> easy to use API (http) > made for big volumes of data

> provided as cloud storage > not relational, schema free

> BASE instead of ACID > no query norm (other than SQL)

**The Cap Theorem**

The CAP theorem dictates, that you can only have

2 of 3 desired traits of a database, these include

**consistency, partition tolerance and availability.**

traditional databases have the **ACID** philosophy which is both

**consistent and tolerant**, however it often blocks transactions.

here comes **NoSQL** with the **BASE** philosophy

it is both available at all times and tolerant,

however it is **NOT consistent during transactions**.

Only after those have stopped will the system get consistent.

> tolerance means the system will work despite partial outage<

### BASE THEOREM

» **Basically Available:**

The database will not block transactions

any and all requests will be responded to (can still fail though!)

» **Soft state:**

The state of the system can change even without input, therefore we consider this to be a soft (not fixed) state

» **Eventual consistency:**

The system will become consistent after end of transactions.

### Key-Value Database

• A simple hash table accessible only through its primary key

• Basically a table with two columns: ID and VALUE

• The value is a blob that the data store just stores: it can be text, JSON, XML, and anything else.

• Operations:

- get the value for the key

- put a value for a key (if the key already exists the corresponding value is overwritten)

- delete a key from the data store

notable examples: Riak, Redis, Memcached, Berkeley DB, HamsterDB, Amazon DynamoDB, Project Voldemort

Key/Value Stores are usually used for this:

» Storing Session Information

» User Profiles, Preferences, Configs

» Shopping carts (LMAO)

They are **NOT** to be used in these cases:

- relationship among data , multioperation transactions

- query by data, operation by sets

### NoSQL Aggregation database

> Arrays (Array Store)

> Dictionaries (Key/Value Stores)

> nested Structures (Document-databases)

### NoSQL Document Store

» Maps a key to a structured Document

» flexible schema , Document stored in JSON or BSON

» Examples: MongoDB, CouchDB

term comparison to RDBMS

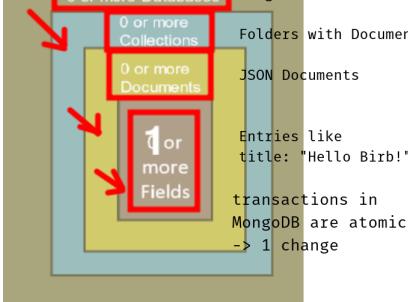
instance -> instance/database, table -> collection

row -> Document , rowid -> \_id/objectId

### MongoDB

» JSON, partitioning via sharding, FOSS :)

» own query language, Document Store (no Schema)



```

var class = {
  _id: ObjectId ("509980df3"),
  course: {code: "Dbs2",
            title: "Advanced DB"},
  year: 2016,
  students: ["Peter", "Manuel", ...],
  no_of_st: 34
}
  
```

- The document contains values of varying types
  - The primary key is \_id and it is of the ObjectId type,
  - The field course is a subdocument, and
  - students is an array of strings

The documents are actually stored as BSON the binary version of JSON with a bit more data. Also there are certain restriction on naming...

### \_id

- The field \_id is reserved for use as a document's primary key

- Its value must be unique in the collection,

- It is immutable,

- May be of any type other than an array or a regular expression type

- MongoDB creates a unique index on the \_id field during the creation of a collection.

- It is always the first field in the documents

- The following are common options for storing values for \_id:

- Use an ObjectId,

- Use a natural unique identifier, if available
  - This saves space and avoids an additional index,

- Generate an auto-incrementing number

- ObjectIds** is a 12-byte BSON type, constructed using:

- A 4-byte value representing the seconds since the Unix epoch,

- A 3-byte machine identifier,

- A 2-byte process id, and

- A 3-byte counter, starting with a random value

- MongoDB uses ObjectIds as the default value for the \_id field if the \_id field is not specified by a client

- Additional benefits of using ObjectIds for the \_id field:

- In the mongo shell, you can access the creation time of the ObjectId, using the `getTimestamp()` method,

- Sorting on ObjectId values is roughly equivalent to sorting by creation time.

### MongoDB Queries

- Expressed via JSON with simple constructs

```
// in orders
```

```
{"orderid":99,
  "customerid":"883c2c5b4e5b",
  "orderDate":"2014-04-23",
  "orderitems":[
    {"product": {"id":27, "name":"NoSQL Distilled"}, "price": 32.45},
    {"product": {"id":55, "name":"Java 4 all"}, "price": 41.33}
  ],
  db.orders.find()
  db.orders.find("customerid":"883c2c5b4e5b")
  db.order.find("customerid":"883c2c5b4e5b"),
  orderid:1, orderDate:1}).db.orders.find("{"orderitems.product.name": "NoSQL"})
```

### sharding

horizontal fragmentation into nodes

reduces IO due to outsourcing to other servers

The shard key -> fragmentation key should be:

- » distributed equally in the data
- » is a field/function or a combination of fields
- should be not unique, but can be a hash on PK?....
- default is hash on \_id aka PK
- shards can usually not be changed -> redo database.

MongoDB only offers few constraints like the unique index constraint.

**advantages:**

- » simple query language, » no schema

**disadvantages:**

- » no constraints

- » no real joins (workaround with lookup)

- » slow/questionable transactions -> BASE

- » security -> default open in the web

### MongoDB Transactions

updates work similar to increasing an array

it even automatically pushes it to a new document if the current one is too big!

however you can avoid relocation by using referencing instead of embedding.

**Performance hindering features:**

- » Atomicity of writes | » Document Growth

- » Sharding | » Indexes and Capped Collections

MongoDB works with **Replica Sets**

child node has the data as well

in case of outage you can still get your data due to the nature of MongoDB this happens without having a specific distributed database.

Also the nodes will pick a new Master if the old one goes down.

**embedded:** ALL IN ONE FILE

Preferred for » ONE-TO-ONE

» ONE-TO-MANY without many overlaps

Warning with this method, uncontrolled growth of document

**referenced:** Stored in multiple files

Preferred when dealing with complex relationships

this is the advantage Document stores have over key/value!

example of reference:

Course Document

```
{
  _id: "Dbs2",
  title: "Advanced Databases",
  coordinator: <ObjectId>,
  guest_lecturer: guest_lect_id,
  year: 2016,
  semester: 5
}

Lecturer Document
{
  _id: <ObjectId>
  name: "Muster",
  email: "pmuster@domain.com"
}

Lecturer Document
{
  _id: <ObjectId>
  name: "Keller",
  teach: ["Dbs1", "Dbs2"],
  email: "sfkeller@hsr.ch"
}
```

MongoDB use one of two methods for relating documents:

» **Manual references** where you save the \_id field of one document in another document as a reference

- These references are simple and sufficient for most use cases

- The other method is to use **DBRefs**

MongoDB documentation recommends using manual references

and here how to create it:

```
use mydb
var coordinator_id = ObjectId()
var guest_lect_id = ObjectId()
```

```
db.class_ref.insert({
  _id: "Dbs2",
  title: "Advanced Databases",
  coordinator: coordinator_id,
  guest_lecturer: guest_lect_id,
  year: 2016,
  semester: 5
})
```

in general, only use references for things that make sense, for example don't make a reference for a blog post or for a persons gender etc. embed the small stuff, reference the big stuff

- MongoDB automatically creates a unique index on the \_id field
- Indexes on fields (other than \_id) that appear often in queries improve performance for common queries
- Indexes are built as BTrees (facilitating range queries)

### createIndex() or ensureIndex()

- db.collection.createIndex(keys, options)

- Parameters:

- keys of the type document:

- For each field to index, a key-value pair with the field and the index order: 1 for ascending or -1 for descending

- options of the type document (optional)

- The most important options:

- unique of the type Boolean:

- The default value is false

- name of the type string:

- If unspecified, MongoDB generates an index name

```
db.collection.ensureIndex(
  {_id: 1, year: -1}, {unique: true}
)
```

### Capped Collection (\*)

- Capped collections are fixed-size collections that support those high-throughput operations that insert and retrieve documents based on insertion order
- Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection

## MongoDB CRUD

### MongoDB Read Operation

- The read operation is defined from within the mongo shell
- It uses `db.collection.find()` method
- The method accepts: selection criteria, projection list, and modifiers as its arguments
- Selection criteria:**
  - Comparison,
  - Existence,
  - Logical junctions (and, or,...),
  - Regular expressions,
  - Array selection objects
- Projection list** contains either a:
  - List of fields to return, or
  - List of fields not to return,
  - The only exception is `_id` field that may be marked as not to return among the list of fields to return
- The read operation returns a cursor to matching documents
- In mongo shell, up to 20 first documents are displayed on the standard output
- Cursor can be used to write handy scripts

### MongoDB Aggregation

- Three ways to do aggregation:
  - Pipeline (**preferred**),
  - Map Reduce, and
  - Single purpose
- Aggregation Pipeline** consists of stages
- Stage operators:**
  - `$match`, `$group`, `$project`, `$sort`, `$skip`...
  - Each stage operator contains expression(s) that specify transformations of documents
  - Expression operators:
    - Boolean, Arithmetic, Comparison, String, Set, Date, Accumulator,...
  - Expressions in the `$group` and `$project` stages use **field path** to access fields in the input documents
  - Examples show use cases

### Lookup mit Aggregation (= aka Joins!)

- Lookup mit Aggregation (= aka Joins!) ab Version 3.2!
 

```
db.absences.aggregate([
  $lookup: {
    from: "holidays",
    pipeline: [
      { $Match: { year: 2018 } },
      { $Project: { _id: 0, date: { name: "$name", date: "$date" } } },
      { $ReplaceRoot: { newRoot: "$date" } },
      { as: "holidays" }
    ]
  }
])
```

### Entspricht Left Join mittels Subquery:

```
SELECT *, holidays
FROM absences
WHERE holidays IN (SELECT name, date FROM holidays
WHERE year = 2018);
```

Siehe MongoDB Docs \$lookup

### MongoDB Write Operation

- There are three classes of write operations in MongoDB:
  - Insert that adds a new document to a collection,
  - Update that modifies an existing document, and
  - Remove that deletes an existing document from a collection.
  - The update and remove operations allow specifying criteria or conditions that identify documents (to be modified, or removed)
    - The syntax of the criteria is the same as in `find()` method
- For all inserts and updates, MongoDB modifies each document in isolation
  - Clients never see documents in an intermediate state
- For multi-document operations, MongoDB does not provide any multi-document transactions or isolation
- The method `db.collection.insert()` writes a document into a collection
  - If there is no `_id` field specified in the document, MongoDB generates it
- The method `db.collection.update()` modifies an existing document, or may insert a new one
- The `update()` method can also rename or delete a field
- The methods `db.collection.remove()` and `drop()` are used to delete just documents or both documents and indices of a collection, respectively

### NoSQL Graph Stores

Made for compact objects with typed aggregations sometimes without schema -> implicit schema or not uniform data

The idea of this database is that everything is an aggregation, -> hence graphs.

Notable examples are: Neo4J, FlockDB, InfiniteGraph biggest gaining database system right now!

Neo4J

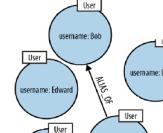
java.... ok fuck this trash

GPL3, embedded, disk based

transaction based on MVCC, scalable

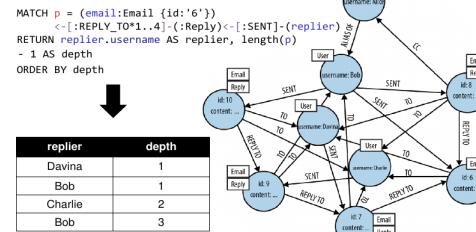
declarative, SQL like, pattern matching (SPARQL)

Collection like Haskell/Python



```
CREATE (alice:User {username:'Alice'}), (bob:User {username:'Bob'}), (charlie:User {username:'Charlie'}), (davina:User {username:'Davina'}), (edward:User {username:'Edward'}), (alice)-[:ALIAS_OF]-(bob), (charlie)-[:User {username:'Charlie'}], (davina)-[:User {username:'Davina'}], (edward)-[:User {username:'Edward'}]
CREATE (bob)-[:EMAILED]-(charlie), (bob)-[:CC]-(davina), (bob)-[:BCC]-(edward)
```

### Path assignment



after we match to a variable called email we check for a chain of "replied\_to" aka we check at least one and max 4 then we get the person who sent the mail and return the depth - 1 also note that the arrow moves to the left meaning the relation is towards the matching node -> away from node <- towards node

### Node:

- » set of labels (usually 1)
- » set of properties

### Edge:

- » Name , » Node->Node direction
- » set of properties

**Property:** key/value of primitive types "name:Dashie"

**types:** Node, Edge, path, map, list, integer, floating point boolean and string

### syntax:

```
(var:Node)-[var:edge]->(var:Node)
(var:Node)<-[var:Edge]->(var:Node)
var may be empty but the ':' has to be written!!
-> [:ALIAS_OF]
you also get the properties of a node by just typing in a variable name -> (varsInsideNode)
could be 1 variable, could be a list
```

**Joins:** Because we can match multiple 'queries' we don't have joins, instead we just

MATCH (query1), (query2) <-Note the comma.

uses for Neo4J: connected data, routing, dispatch, location based services and recommendation engines.

when not to use: few edges/interactions, update on all nodes big data... ? nice word

### NoSQL Column 'Family' store

These databases store a full row, within a single cell of a column

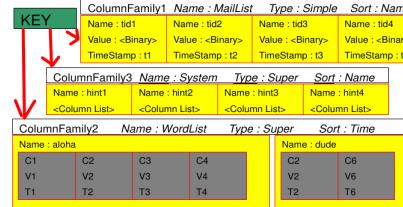
[1 | Dashie | 85 | f] -> [1|name:Dashie level:85 sex:f the | symbol is to separate columns

Notable column family stores:

BigTable (Google), HBase (Apache), Hypertable, Cassandra (Apache) PNUTS (Yahoo)

### Cassandra

- » Keyspace: aka database within a Node
- » Node: data storage -> like document for MongoDB
- » Data Center: Collection of nodes
- » Cluster: collection of data centers



### Cassandra Column

- Column: basic unit of storage in Cassandra
- A name-value pair; always stored with a timestamp (used to resolve write conflicts, deal with stale data, etc.)
- Example of a "simple column":

```
{
  name: "fullName",
  value: "Martin Fowler",
  timestamp: 12345667890
}
```

### Cassandra "Column Family"

- A row is a collection of columns linked to a key
- A "column family" is a collection of grouped rows

### Example:

//column family

```
{ /row
  "pramod-sadalage" : {
    firstName: "Pramod",
    lastName: "Sadalage",
    lastVisit: "2012/12/12"
  },
  /row
  "martin-fowler" : {
    firstName: "Martin",
    lastName: "Fowler",
    location: "Boston"
  }
}
```

### Partitioning

• Consistent Hashing Algorithm

• Logical Ring of hash values

• Each node is given a position on this ring

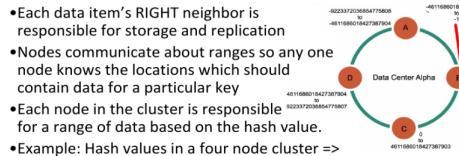
• Each node is responsible for a LEFT range of hashes

• Each data item's RIGHT neighbor is responsible for storage and replication

• Nodes communicate about ranges so any one node knows the locations which should contain data for a particular key

• Each node in the cluster is responsible for a range of data based on the hash value.

• Example: Hash values in a four node cluster =>



### Indexing

You can index columns other than the keys for the column family

UPDATE COLUMN\_FAMILY Customer

WITH comparator = UTF8Type

AND column\_metadata = [{

column\_name: city,

validation\_class: UTF8Type,

index\_type: KEYS}];

GET Customer WHERE city = 'Boston';

### Cassandra Query Language (CQL)

• CREATE COLUMNFAMILY Customer (

KEY varchar PRIMARY KEY,

name varchar,

city varchar,

web varchar);

• INSERT INTO Customer (KEY,name,city,web) VALUES

('mfowler', 'Martin Fowler', 'Boston', 'www.martinfowler.com');

• SELECT \* FROM Customer;

SELECT name, web FROM Customer;

SELECT name, web FROM Customer WHERE city='Boston';

### Use cases:

» Event logging (ex: error logging)

good since cassandra scales with writes.

» Content management systems, Blogging platforms

-> good for handling post entries.

» counters in web applications (ex: count visitors)

### when not to use:

» ACID requirement

» no aggregate functions available

» prototypes, query changes can take time.

### Column Store

#### OLTP: Online Transaction Processing

» low data volume | » High volume of transactions

» Typically normalized data | » ACID compliance

» high availability || » relational db (pgsql)

» data management optimized

#### OLAP: Online Analytical Processing

» high data volume | » low transaction volume

» denormalized data | » **not necessarily** ACID compliant

» usually don't need high availability

» read optimized system

Usually these two are separated, but recently there have been efforts to combine those two just like BASE and ACID are getting closer.

### Customers Table Row Store

ID	Name	Address	City	State	Bal Due
1	Bob	---	---	---	3,000
2	Jim	---	---	---	500
3	---	---	---	---	3,700
4	---	---	---	---	1,500
5	Liz	---	---	---	0
6	Dave	---	---	---	9,000
7	Sue	---	---	---	1,010
8	Bob	---	---	---	50
9	Jim	---	---	---	1,300

### Column Store

ID	Name	Address	City	State	Bal Due
1	Bob	---	---	---	3,000
2	Jim	---	---	---	500
3	---	---	---	---	3,700
4	---	---	---	---	1,500
5	Liz	---	---	---	0
6	Dave	---	---	---	9,000
7	Sue	---	---	---	1,010
8	Bob	---	---	---	50
9	Jim	---	---	---	1,300

### encoding and compression

in databases it is the default to compress data when you have high selectivity

-> male / female  
instead of storing male/female in every column or row we can instead compress this attribute in order to save space

### Run Length Encoding (RLE)

Querier	Male	Female	Row
Q1	1	330	100
Q2	311	280	335
Q3	311	100	100
Predict	1	5	315
Start	2	6	450
Count	51	5	320
Q4	2	56	3
End	51	5	150
Q5	1	5	256
Q6	1	5	192
Q7	1	5	184
Q8	1	5	310
Q9	1	5	251
Q10	1	5	266

the way this is done is by storing the possibilities, aka/male/female in another table.

This means that we can then reference the id of male or female. Note that this is all done implicitly, nothing has to be defined.

### Column store advantages:

» better compression | » less disk usage

» automatic indexing | » late materialization

Column stores disadvantages:

» bad writes | » !!! don't Select \* from !!! | » OLAP

### Row stores advantages:

» easy partitioning and parallelization

» good writes | » seq row scans

### Row stores disadvantages:

» column read == read all rows... | » more storage

» early materialization | OLTP

**late vs early materialization**  
early means that everything has to be read first while late means that we wait until the end of the query. return just the column with nothing else.

### NoSQL In Memory Stores (IMDB)

this database stores its values **inside the RAM** However, it still stores **regular Snapshots** of its system on the regular disk, in order to prevent data loss.

It also creates logs, to restore previous states in case of a shutdown. Small note: the **pagefile** is also loaded from the disk on startup

**advantages:** » fast read | » ACID

» indexing on the fly

» no buffer for write to disk -> happens with snapshots

less power requirement due to less software

**disadvantages:** » small storage...

» the values are compressed to save storage -> slow writes

## MonetDB and NoSQL end

IMDB examples: SAP HANNA, Oracle TimesTen, IBM SolDB, MonetDB

» Column oriented Database | » FOSS, written in C » SQL,C,R scripts on database itself  
» supports (application): C,C++,Java,Javascript, Perl,PHP,Python,R,Rust  
» used for Data mining, analysis and text search  
» every column mapped to 1 file | » columns are compressed  
» columns are automatically indexed  
» multi-core parallel execution of queries  
» data-manipulation language SQL 2003, sql Explain, Xquery XML  
» OLAP | » no secondary indexes (only automatic ones)  
» ACID/Optimistic Concurrency Control -> OCC  
» 100x faster than psql in TPC-H benchmark -> readonly

so then, why not use NoSQL all the time?

!NoSQL lacks multimodel databases!

always just one -> document, key/value, graph, etc

Database benchmarking: » TPC.org, pgbench (pgsql)

» timing | » data usage | » query types | » data amount..

execution models:

» volcano execution model: row by row -> ex:pgsql

vectorized execution model: value by value -> MonetDB

## Database as a Service (DBaaS)

Providers: Amazon AWS, Microsoft Azure, Google Cloud, ..

## Pagination Paging

this is essentially just an offset -> where x > 10000

## GraphQL

declarative query language over the web

schemas with types | » description of data via API

» one endpoint for interaction

» easy to develop API | » clients can request data

» Query and Data manipulation language (DML)

» Data-Type-System (DDL)

» supported by all big databases (pgsql)

» results represent JSON docs but uses edges and nodes

In the end GraphQL handles input from the user

towards a database like PSQL or others

it is in itself not a database!!

## GraphQL: Schema Definition Language (SDL)

• Datentypen «Scalar Types»:

- Int (32-bit integer); Float (signed double-precision); String (UTF-8); Boolean
- ID: unique identifier
- Custom scalar types, e.g. Date type: scalar Date

• Weitere Typen: Enum, List, Union, Interface, ...

• Schema:

- «Object type» mit «Fields»: Beispiel type abteilung { abtnr: Int! name: String }
- „Query type“ (=READ) und optional je einen „Mutation type“ für CREATE, UPDATE und DELETE, - die «Input Types» verlangen

**schema-first vs code-first:**

schema first allows for better and faster integration

however it is obviously more complicated to execute.

## PostGraphile: GraphQL and PSQL

» library for integrating PSQL into GraphQL

» PSQL database, NodeJS http-server, GraphQL webAPI

» integrates GraphiQL (GUI tool to write queries)

schema first!

```
CREATE TABLE app_public.users (
    id serial PRIMARY KEY,
    username citext NOT NULL unique,
    name text NOT NULL,
    about text,
    organization_id int NOT NULL
    REFERENCES app_public.organizations ON DELETE CASCADE,
    is_admin boolean NOT NULL DEFAULT false,
    created_at timestamp NOT NULL DEFAULT now(),
    updated_at timestamp NOT NULL DEFAULT now()
);
```

table: create type -> User (CamelUpperCase)

attributes: organization\_id,name,about -> organizationId (camelCase)

creates nodeId if PK exists

creates aggregationField (variable name) for FK -> userByOrganizationId

it also creates createUser, updateUser,deleteUser for CRUD compliance

for queries:

tablename: app\_public.users -> allUsers name from above with all and s

unique constraints: for all unique constraint attributes it will create:

name -> userByName

a foo(nodeId: ID!) is obligatory -> ! denotes obligatory

and here the query as an example:

```
# "query" kann weggelassen werden:
query {
    allAngestelltenList(first: 5, offset: 0) {
        "data": {
            allAngestelltenList: [
                {
                    "name": "Marxer, Markus",
                    "persnr": 1001,
                    "abtnr": 1,
                    "chef": null
                },
                {
                    "name": "Widmer, Anna",
                    ...
                }
            ]
        }
    }
}

query {
    allAngestellten {
        totalCount: # Info
        pageInfo: # Paginierung
        startCursor: ...
        endCursor: ...
    }
    nodes: [
        {
            "name": "Marxer, Markus",
            "persnr": 1001,
            "abtnr": 1,
            "chef": null
        }
    ]
}
```

» the default creation of createUser etc can be disabled you can also redefine what the table returns on query ex: user , userByRowId, allUsers, allUsersList keep in mind that the first 2 return 1 user the other return a list

the third one can also be used to

get more info about the nodes and edges see "paginierung" above"

**best practices:** » don't create nullable fields

» use mutations -> convert long types to mutations

GraphQL vs SQL » GraphQL is very limited

» controlled | » expandable

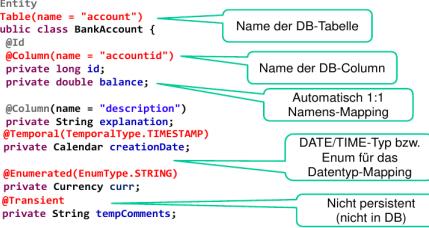
here an example for CRUD functions

```
mutation MyDeleteAbteilung99 {
    deleteAbteilungByAbtnr(input: { abtnr: 99 }) {
        # clientMutationId
        # deletedAbteilungId
        abteilung {
            abtnr
            name
        }
    }
}
```

Legende: rot=Metadaten (hier auskommentiert); grün: Rückgabedaten

## OR MAPPER

Application -> Persistence API -> JPA Provider -> JDBC API -> JDBC Driver -> Database  
» start with @Entity | » inheritance and interfaces possible  
» can be abstract | » no constructor arguments (unless private)  
» needs primary key | » class may not be final  
» fields should be private or protected | » Read committed!



the red line are optional, if the names in the java object and in the db are the same !!  
supported datatypes: primitive types, wrapper types strings, enums, date, serializable classes  
supported relations: collection, set, map of entities  
execute statements

**Base:** Name der Persistence Unit in persistence.xml  
EntityManagerFactory factory = Persistence.createEntityManagerFactory("Bank"); EntityManager em = factory.createEntityManager();

**Action or query** insert / delete  
em.close(); update / query  
!this is the base, we always need to call this!! note that entity managers are independent comparing the same result from 2 entity managers will not be considered equal

**insert**  
BankAccount account = new BankAccount(); account.setName("Bill"); em.persist(account); em.getTransaction().commit();

**delete**  
BankAccount account = em.find(BankAccount.class, 1L); account.incBalance(100);

**update**  
BankAccount account = em.find(BankAccount.class, 1L); em.remove(account);

**query**  
Query query = em.createQuery("SELECT a FROM BankAccount a"); List<BankAccount> list = query.getResultList(); for (BankAccount account : list) { System.out.println(account); }

further type details can be specified as such:

```
@Column(name="ssn", unique=true, nullable=true)
private long ssn;
@Column(name="l_name", length=200)
private String lastName;
@Column(name="salary", scale=10, precision=2)
private BigDecimal salary;
@Column(name="s_time", columnDefinition="TIMESTAMPTZ")
private Calendar startDate;
```

## Field Access & Property Access

in OR mapping you don't need to specify setId or get Id they are automatically created. -> Field Access Should you still create them -> Property Access for field access the @Id must be above the variable declaration for the property access must to be above the getID() function.

## order of mapping:

XML / user defined mapping / default mapping

## One-To-One

```
BankCustomer 1..1 0..1 Address
@entity
@Table(name = "bankcustomer") default true
public class BankCustomer {
    @OneToOne(optional = true) Nullable
    @JoinColumn(name = "customer_addressid") // FK
    private Address address;
```

## Bi-One-To-One

this is an optional ADDITION to above it just creates a manual mapping for the bankcustomer

```
@entity Bi == Bidirectional
@Table(name = "address")
public class Address {
    @OneToOne(mappedBy = "address", optional = false)
    private BankCustomer customer;
```

## Inverse One-To-One

```
BankCustomer 0..1 1..1 Address
@Entity
@Table(name = "address")
public class Address {
    @OneToOne(name = "addressid", @Id
    referencedColumnName = "customer_addressid", insertable = false, updatable = false)
    private BankCustomer customer;
```

For all many to x

private Collection<BankAccount> accounts = new ...

instead of private something = new ..

Many-To-One -> change the @OneToOne with @ManyToOne

Bi-Many-To-One -> change the second @OneToOne with @OneToMany

## Many-To-Many

```
BankManager 0..* 0..* BankCustomer
```

```
@Entity
@Table(name = "bankmanager")
class BankManager {
    @ManyToMany
    @JoinTable(name = "customer_manager",
    joinColumns = {@JoinColumn(name = "managerref")},
    inverseJoinColumns = {@JoinColumn(name = "customerref")})
    private Collection<BankCustomer> customers = ...}
```

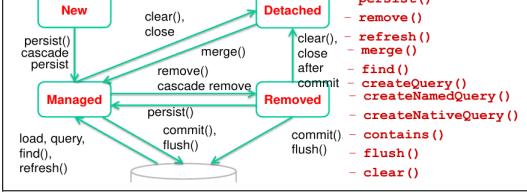
Bi-Many-To-Many -> Bi-OneToOne with @ManyToMany

## Fetch Types

Eager -> default OneToOne/ManyToOne entity loaded directly Lazy -> default OneToMany/ManyToMany load only on access @relation(fetch = FetchType.LAZY/EAGER for override)

## O/R Mapping Variants

» Top Down Code first | » Bottom up DB first  
» Inside-out Mapping first | » Outside in DB Code together



## Generated Ids

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long accountId;
```

This is the same as SERIAL in SQL

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "BankCustGen")
@SequenceGenerator(name = "BankCustGen",
sequenceName = "customeridseq",
allocationSize=1) // nur einmal ausführen
```

private long customerId;

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
generator = "CustomerGen")
```

```
@TableGenerator(name = "CustomerGen", table = "keytable",
pkColumnName = "keyname",
valueColumnName = "keyvalue",
pkColumnValue = "customerkey")
```

private long customerId;

There is also the option for cascade, both persist and delete

@OneToMany(cascade = CascadeType.PERSIST, ...) CascadeType.REMOVE)

## Aggregation and Composition

```
BankCustomer 0..* BankAccount
```

@OneToMany(cascade = CascadeType.PERSIST, ...) private Collection<BankAccount> accounts = new ArrayList<>();

```
BankCustomer 0..* BankAccount
```

@OneToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE }, ...) private Collection<BankAccount> accounts = new ArrayList<>();

for bidirectional relation, MANUALLY sync ANY updates, they are NOT automatic!!

## inheritance strategy

Single Table

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "type")

public abstract class BankCustomer {

@Id private String name;

Joined Table

@Entity

@Inheritance(strategy = InheritanceType.JOINED)

@DiscriminatorColumn(name = "type")

public abstract class BankCustomer {

@Id private int customerId;

private String name;

Multi Table

@Entity

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

public abstract class BankCustomer {

@Id private int customerId;

private String name;

child tables (multi table without discriminator)

@Entity

@DiscriminatorValue("Private")

public class PrivateBankCustomer extends BankCustomer {

private String eliteOffer;

## JPQL Query language optimized for OR

essentially the same other than this

### Positional Parameters

select a from Bankaccount where a.customer.name like ?1 and a.balance > ?2

### Named Parameters

select a from Bankaccount a where a.customer.name like :name and a.balance > :lower

### dynamic queries:

```
Query query = em.createQuery(
"SELECT c FROM BankCustomer c WHERE c.name LIKE :customerName"
);
query.setParameter("customerName", name);
query.setMaxResults(1000);
List<BankCustomer> list = query.getResultList();
```

### Predefined query

```
@NamedQuery(name = "CustomerSearch", query =
"SELECT c FROM BankCustomer c WHERE c.name LIKE :customerName")
@Entity
public class BankCustomer { ... }
Query query = em.createNamedQuery("CustomerSearch");
query.setParameter("customerName", name);
```

Locktype: either pessimistic or optimistic

BankAccount account = em.find(...);

em.lock(account, LockModeType.PESSIMISTIC\_WRITE);

account.incBalance(100);

merge, add to existing or create new

public Customer storeUpdatedCustomer(Customer cust) { return entityManager.merge(cust); }

refresh entity manager

em.refresh(customer);

JSON to PSQL in separate file. OPTIONAL!