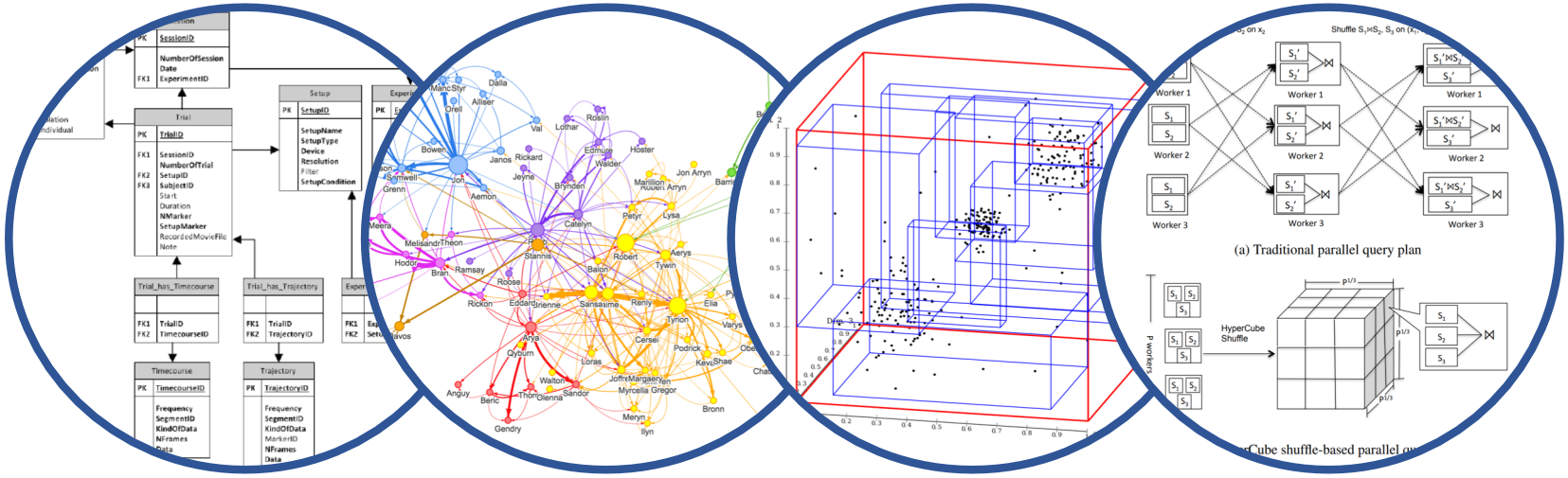


Introduction to Data Management

SQL++ Nesting

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Announcements

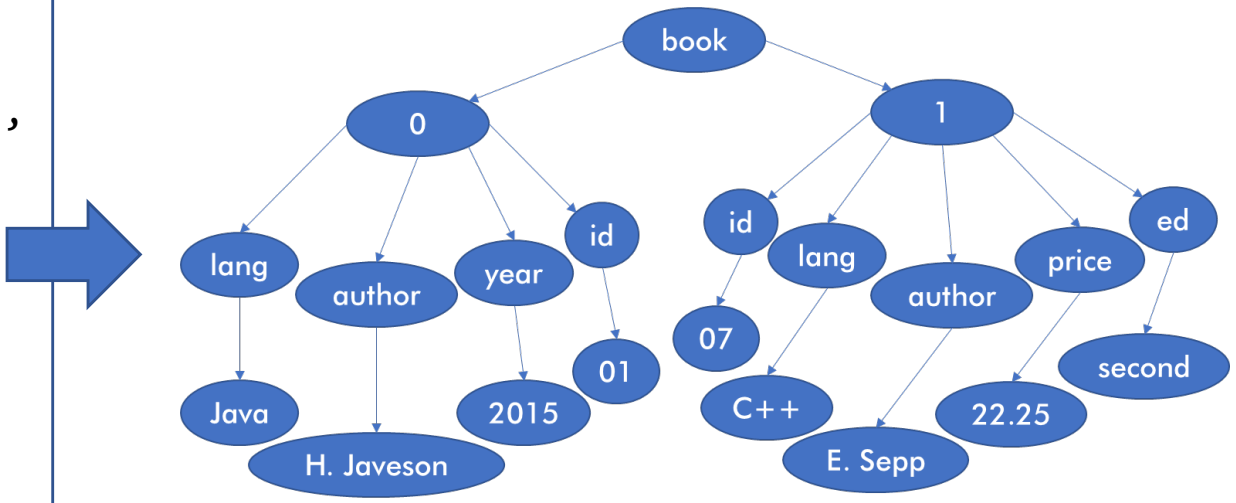
- Final exam topics:
  - Technically all material, but strong focus on post-midterm material. Still need to write SQL queries and read RA trees.
  - No cost estimation (I/Os from RA trees)
  - Yes cardinality estimation: given query and table statistics, how many tuples do we expect?
  - No writing Java code. Map-Reduce answers can be in pseudo-code

- From HW 6: Make sure to shut down all spark clusters!! Check EMR on every region.

Recap: Semi-Structured Data Key Features

- Tree-like data
- Embedded schema

```
{
   "book":[
      {
         "id": "01",
         "language": "Java",
         "author": "H. Javeson",
         "year": 2015
      },
      {
         "author": "E. Sepp",
         "id": "07",
         "language": "C++",
         "edition": "second",
         "price": 22.25
      }
   ]
}
```



Recap: JSON and ADM

- AsterixDB uses a JSON-like encoding called ADM
  - Multisets
  - uuids
- SQL++ queries work on arrays and multisets like SQL queries work on tables

```
SELECT x.phone
  FROM [
         {"name": "Dan", "phone": [300, 150]},
         {"name": "Alvin", "phone": 420}
       ] AS x;

-- output, same for-loop semantics like in SQL
/*
{ "phone": [300, 150] }
{ "phone": 420 }
*/
```

Today

Last time:

- The Asterix Data Model (ADM)

Today:

- SQL++ crash course
  - Data Definition Language (DDL)
    - Defining structure beyond self-description
    - Indexing
  - Data Manipulation Language (DML)
    - Joins
    - Nesting and Unnesting

DDL? DML?

- You have seen it all before!

	SQL Examples	SQL++ Examples
Data Description Language (DDL)	CREATE DATABASE... CREATE TABLE... CREATE INDEX... DROP TABLE... ALTER TABLE... (unique)	CREATE DATAVERSE... CREATE TYPE... (unique) CREATE DATASET... CREATE INDEX... DROP DATASET...
Data Manipulation Language (DML)	SELECT...FROM... INSERT INTO... DELETE FROM...	SELECT...FROM... INSERT INTO... DELETE FROM...

DDL? DML? What the hell?

- You have seen it all before!

	SQL Examples	SQL++ Examples
Data Description Language (DDL)	CREATE DATABASE C C D A	CREATE DATAVERSE
Data Manipulation Language (DML)	SELECT…FROM… INSERT INTO… DELETE FROM…	SELECT…FROM… INSERT INTO… DELETE FROM…

Schema Manipulation

DDL? DML? What the hell?

- You have seen it all before!

	SQL Examples	SQL++ Examples
Data Description Language (DDL)	CREATE DATABASE C C D A	CREATE DATAVERSE
Data Manipulation Language (DML)	SELECT … FROM I D	SELECT … FROM

Schema Manipulation

Data Manipulation

Today

Today:

- SQL++ crash course
  - **Data Definition Language (DDL)**
    - Defining structure beyond self-description
    - Indexing
  - Data Manipulation Language (DML)
    - Joins
    - Nesting and Unnesting

Data Definition Language (DDL)

- Didn't we say that the schema is already embedded in the data?

- Opportunity to give definitions to objects
  - Ad hoc querying possible but not optimal
  - More structure → **Better defined application**
  - More structure → **Better performing queries**

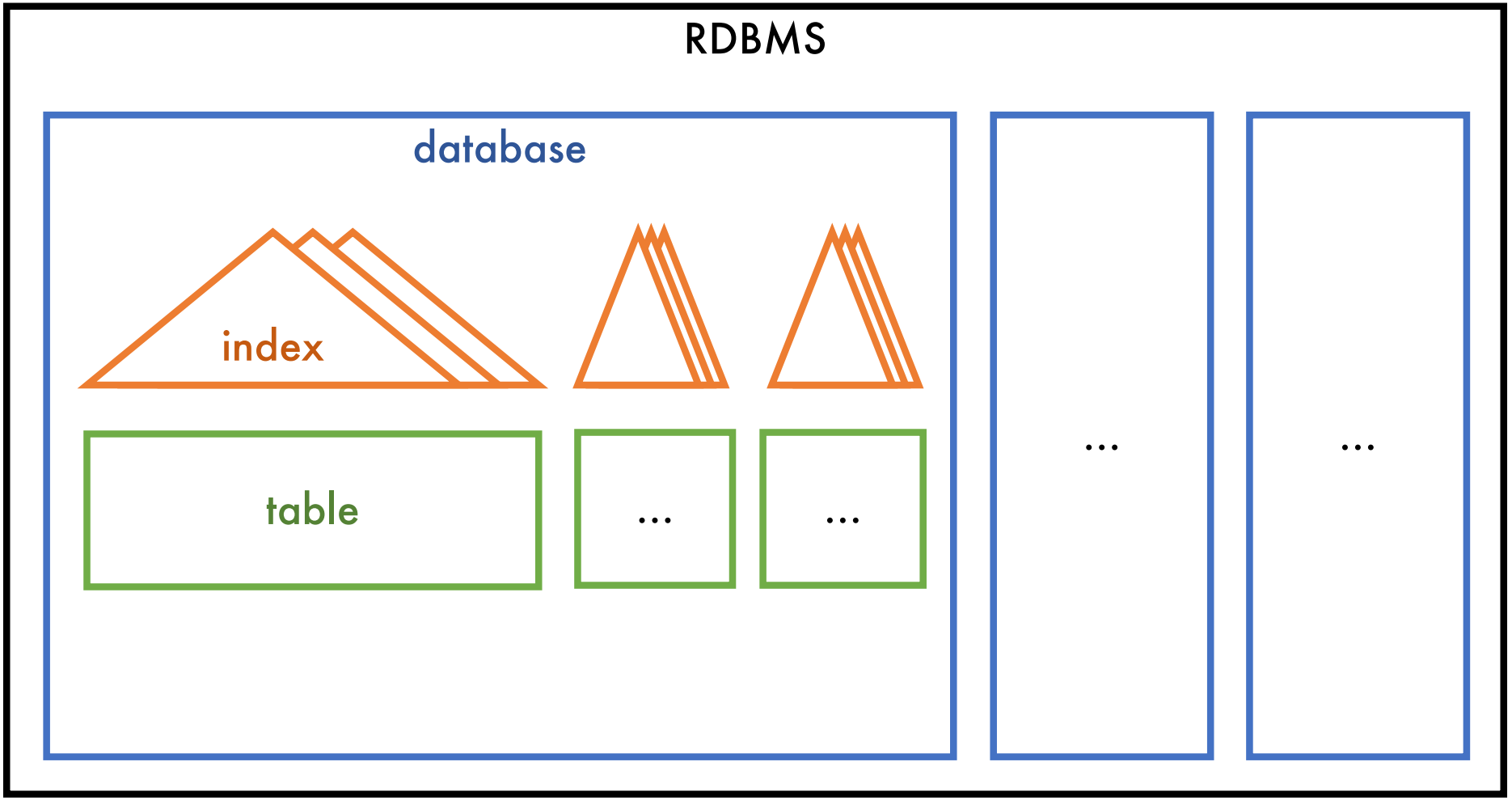
  - Remember from last time:

```
SELECT x.fone -- intentional typo but no error
```

  - Data definition helps us catch these

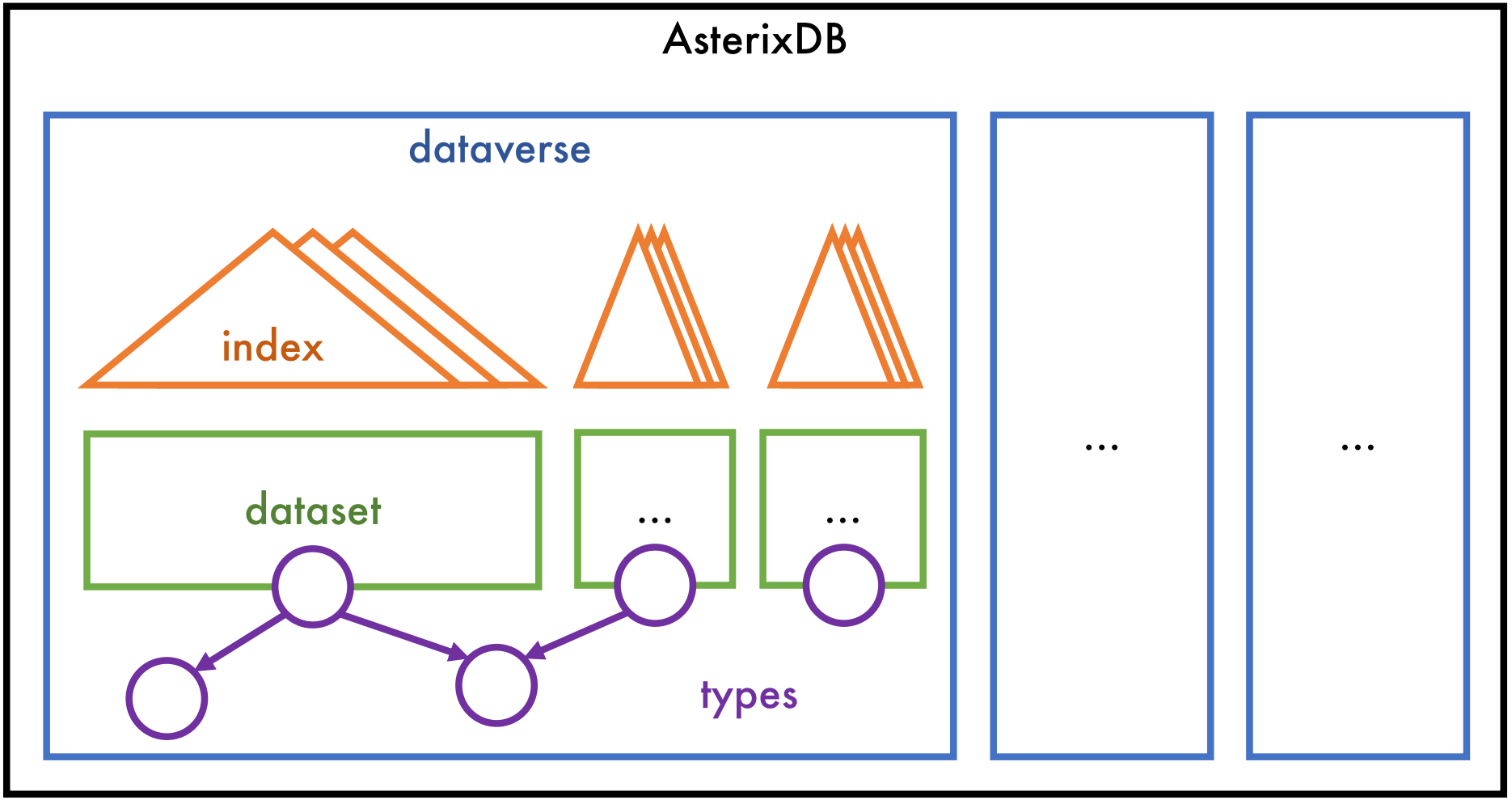
Data Definition Language (DDL)

- Extremely similar to the relational world



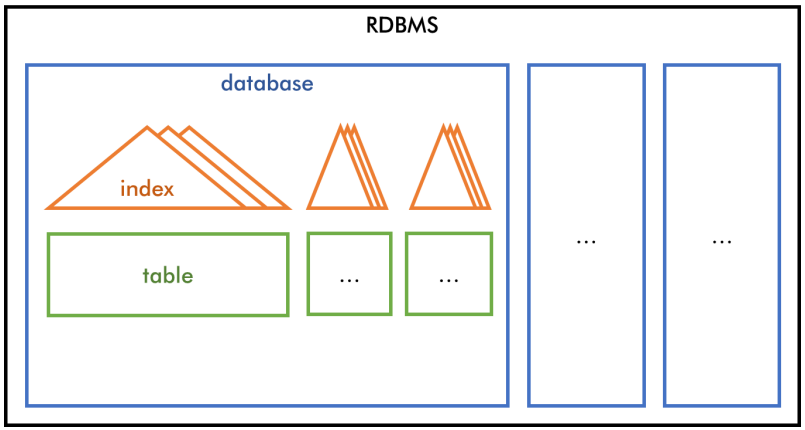
Data Definition Language (DDL)

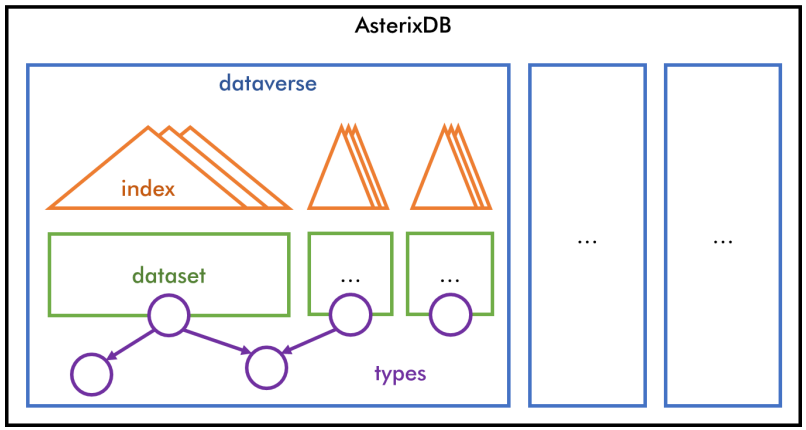
- Extremely similar to the relational world



Data Definition Language (DDL)

- Extremely similar to the relational world





Functionality	RDBMS	AsterixDB
Namespace	Database	Dataverse
Data Collection	Table	Dataset
Data Access	Index	Index

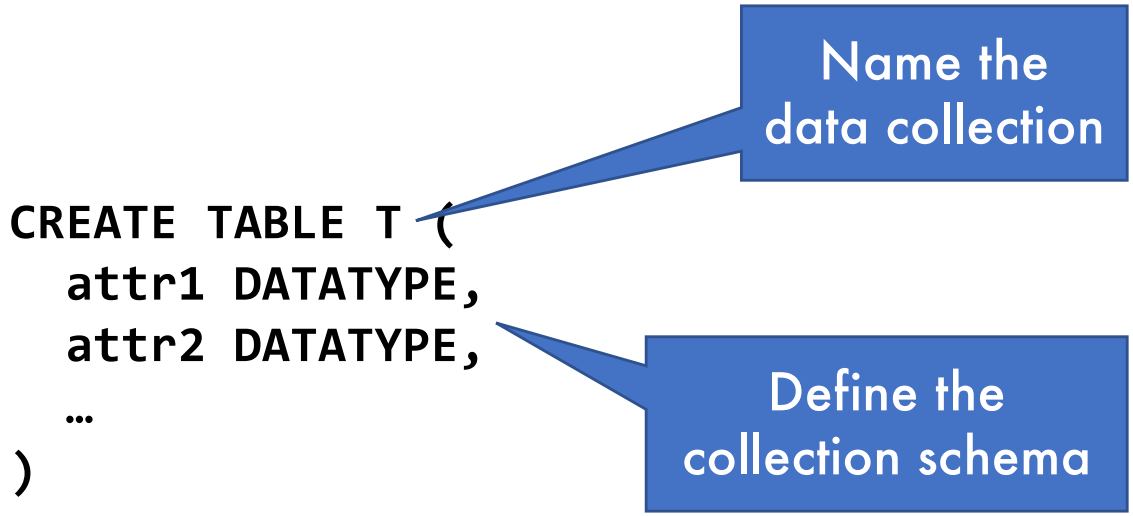
Types

What is this SQL statement doing?

```
CREATE TABLE T (
  attr1 DATATYPE,
  attr2 DATATYPE,
  …
)
```

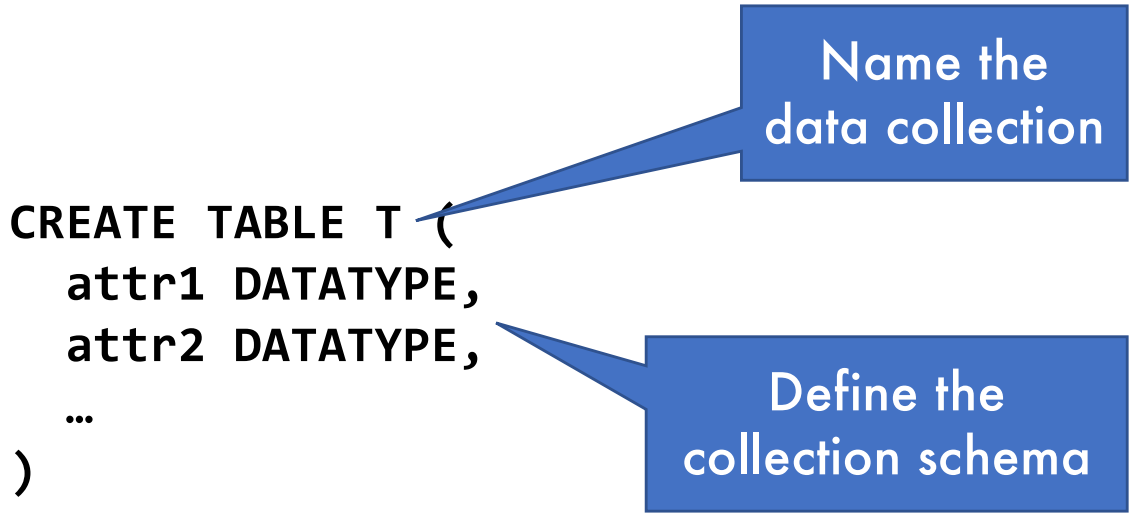
Types

What is this SQL statement doing?



Types

What is this SQL statement doing?



Flat data can do it all in one step!
What about nested data?

Types

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567",
         "orders": [
            {
               "date": 1997,
               "product": "Furby"
            }
         ]
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678",
         "orders": [
            {
               "date": 2000,
               "product": "Furby"
            },
            {
               "date": 2012,
               "product": "Magic8"
            }
         ]
      },
      {
         "name": "Magda",
         "phone": "555-345-6789",
         "orders": []
      }
   ]
}
```

Types

Need to describe person schema

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567",
         "orders": [
            {
               "date": 1997,
               "product": "Furby"
            }
         ]
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678",
         "orders": [
            {
               "date": 2000,
               "product": "Furby"
            },
            {
               "date": 2012,
               "product": "Magic8"
            }
         ]
      },
      {
         "name": "Magda",
         "phone": "555-345-6789",
         "orders": []
      }
   ]
}
```

Types

Need to describe person schema

Person schema needs orders schema!

```
{
   "person":[
      {
         "name": "Dan",
         "phone": "555-123-4567",
         "orders": [
            {
               "date": 1997,
               "product": "Furby"
            }
         ]
      },
      {
         "name": "Alvin",
         "phone": "555-234-5678",
         "orders": [
            {
               "date": 2000,
               "product": "Furby"
            },
            {
               "date": 2012,
               "product": "Magic8"
            }
         ]
      },
      {
         "name": "Magda",
         "phone": "555-345-6789",
         "orders": []
      }
   ]
}
```

Types

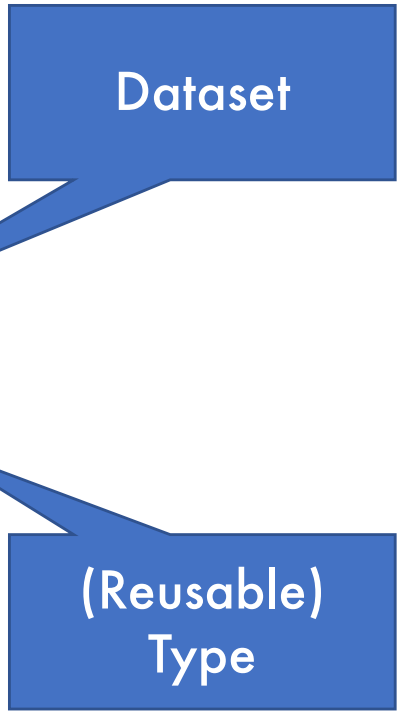
Less abstraction!

Need a way to specify
top-level collection
in addition to
general collection schema

Types

Less abstraction!

Need a way to specify
top-level collection
in addition to
general collection schema



Types

```
CREATE TABLE T (
  attr1 DATATYPE,
  attr2 DATATYPE,
  …
)
```

Name the data collection

Define the collection schema

Less abstraction!

Need a way to specify
top-level collection
in addition to
general collection schema

Dataset

(Reusable) Type

Types

- **Types define the schema of some collection** (not necessarily a top-level one)

- How to:
  - List all **required** fields
  - List all **optional** fields with "?" (can be missing)
  - Specify **CLOSED/OPEN**
    - CLOSED → no other fields except the listed ones are allowed
    - OPEN → extra fields (not listed) are allowed

Closed Types

- Strict adherence to schema (no additional fields)

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan",
      "phone": 5551234567,
      "email": "suciu@cs"
   },
   {
      "name": "Alvin",
      "phone": 5552345678,
      "email": "akcheung@cs"
   }
]
```

Closed Types

- Strict adherence to schema (no additional fields)

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan",
      "phone": 5551234567
   },
   {
      "name": "Alvin",
      "phone": 5552345678
   }
]
```

Closed Types

- Strict adherence to schema (no additional fields)

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan",
      "phone": 5551234567,
      "email": "suciu@cs"
   },
   {
      "name": "Alvin",
      "phone": 5552345678
   }
]
```

Closed Types

- Strict adherence to schema (no additional fields)

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan"
   },
   {
      "name": "Alvin",
      "phone": 5552345678
   }
]
```

Can't be missing required fields

Closed Types

- Strict adherence to schema (no additional fields)

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan",
      "phone": 5551234567,
      "email": "suciu@cs"
   },
   {
      "name": "Alvin",
      "phone": 5552345678,
      "likesBananas": true
   }
]
```

Can't use unspecified fields

Open Types

- Allows additional fields

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan",
      "phone": 5551234567,
      "email": "suciu@cs"
   },
   {
      "name": "Alvin",
      "phone": 5552345678,
      "likesBananas": true
   }
]
```

Open Types

- Allows additional fields

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
  name:  string,
  phone: int,
  email: string?
}
```

```
[
   {
      "name": "Dan"
   },
   {
      "name": "Alvin",
      "phone": 5552345678
   }
]
```

Same as before

Collection Data Types

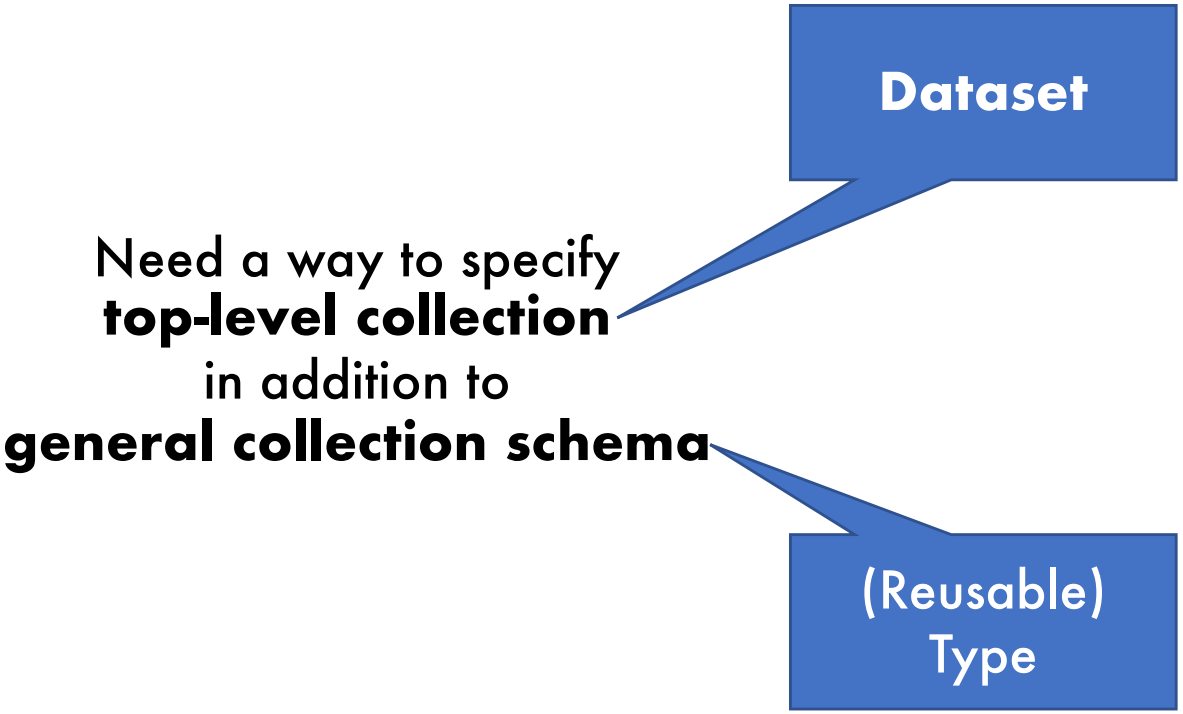
- Data can be a collection

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: [int]
}
```

Mean phone is an array of ints

```
[
   {
      "name": "Dan",
      "phone": [5551234567]
   },
   {
      "name": "Alvin",
      "phone": [5552345678, 5553456789]
   },
   {
      "name": "Magda",
      "phone": []
   }
]
```

Datasets


Dataset
Need a way to specify
top-level collection
in addition to
general collection schema
(Reusable)
Type

Dataset Keys

- Must be present for a dataset
  - For lookup ability
  - Secondary indexing
  - Sharding/Partitioning

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int
}

DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
  PRIMARY KEY name;
```

Dataset Keys

- Must be present for a dataset
  - For lookup ability
  - Secondary indexing
  - Sharding/Partitioning

What if there are no good keys?

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int
}

DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
  PRIMARY KEY name;
```

Dataset Keys

- Must be present for a dataset
  - For lookup ability
  - Secondary indexing
  - Sharding/Partitioning

What if there are no good keys?

Autogenerate!

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int
}

DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
  PRIMARY KEY myKey AUTOGENERATED;
```

Dataset Keys

- Must be present for a dataset
  - For lookup ability
  - Secondary indexing
  - Sharding/Partitioning

What if there are no good keys?

Autogenerate!

```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
  name:  string,
  phone: int
}

DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
  PRIMARY KEY myKey AUTOGENERATED;
```

Each object will have a uuid field named "myKey"

Today

Today:

- SQL++ crash course
  - Data Definition Language (DDL)
    - Defining structure beyond self-description
    - Indexing
  - **Data Manipulation Language (DML)**
    - Joins
    - Nesting and Unnesting

Joins

- Same nested-loop semantics as SQL!

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p, Orders AS o
 WHERE p.name = o.pname;
```

```
for each object in p:
    for each object in o:
        if WHERE satisfied:
            …
```

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567"
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678"
   },
   {
      "name": "Magda",
      "phone": "555-345-6789"
   }
}}
```

```
-- Dataset Orders
{{
   {
      "pname": "Dan"
      "date": 1997,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2000,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2012,
      "product": "Magic8"
   }
}}
```

SQL++ Aggregation

Better encapsulation of 3-valued logic!

	Function	NULL	MISSING	Empty Collection
NULL considered	STRICT_COUNT	counted	counted	0
	STRICT_SUM	returns NULL	returns NULL	returns NULL
	STRICT_MAX	returns NULL	returns NULL	returns NULL
	STRICT_MIN	returns NULL	returns NULL	returns NULL
	STRICT_AVG	returns NULL	returns NULL	returns NULL
NULL ignored (same as vanilla SQL)	ARRAY_COUNT	not counted	not counted	0
	ARRAY_SUM	ignores NULL	ignores NULL	returns NULL
	ARRAY_MAX	ignores NULL	ignores NULL	returns NULL
	ARRAY_MIN	ignores NULL	ignores NULL	returns NULL
	ARRAY_AVG	ignores NULL	ignores NULL	returns NULL

SQL++ Aggregation

Better encapsulation of 3-valued logic!

NULL

Use this "array_count(x)"

NULL ignored (same as vanilla SQL)

Function	NULL	MISSING	Empty Collection
STRICT_COUNT	counted	counted	0
STRICT_SUM	returns NULL	returns NULL	returns NULL
RICT_MAX	returns NULL	returns NULL	returns NULL
RICT_MIN	returns NULL	returns NULL	returns NULL
TRICT_AVG	returns NULL	returns NULL	returns NULL
ARRAY_COUNT	not counted	not counted	0
ARRAY_SUM	ignores NULL	ignores NULL	returns NULL
ARRAY_MAX	ignores NULL	ignores NULL	returns NULL
ARRAY_MIN	ignores NULL	ignores NULL	returns NULL
ARRAY_AVG	ignores NULL	ignores NULL	returns NULL

Nested Data

- Two interesting directions
  - Nested data → Unnested results
  - Unnested data → Nested results

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

```
SELECT p.name, p.phone,
       p.orders.date, p.orders.product
  FROM Person AS p;

-- ERROR
```

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

```
SELECT p.name, p.phone,
       p.orders.date, p.orders.product
  FROM Person AS p;

-- ERROR
```

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

```
SELECT p.name, p.phone,
       p.orders.date, p.orders.product
  FROM Person AS p;

-- ERROR
```

Dereferencing can only be done on objects!

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p, p.orders AS o;

-- output
/*
{name: Dan,   phone: 555-123-4567, date: 1997, product: Furby}
{name: Alvin, phone: 555-234-5678, date: 2000, product: Furby}
{name: Alvin, phone: 555-234-5678, date: 2012, product: Magic8}
*/
```

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p UNNEST p.orders AS o;

-- output
/*
{name: Dan,   phone: 555-123-4567, date: 1997, product: Furby}
{name: Alvin, phone: 555-234-5678, date: 2000, product: Furby}
{name: Alvin, phone: 555-234-5678, date: 2012, product: Magic8}
*/
```

Same semantics as ","

Nested Data → Unnested Results

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": [
         {
            "date": 1997,
            "product": "Furby"
         }
      ]
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- How do we **unnest** data?
  - SQL++ can unnest and join all at once (built into syntax)

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p UNNEST p.orders AS o;

-- output
/*
{name: Dan,   phone: 555-123-4567, date: 1997, product: Furby}
{name: Alvin, phone: 555-234-5678, date: 2000, product: Furby}
{name: Alvin, phone: 555-234-5678, date: 2012, product: Magic8}
*/
```

Parent-child join!

Unnesting Non-Uniform Data

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": {
         "date": 1997,
         "product": "Furby"
      }
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678"
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

object

array

- What if data is not uniform?

Unnesting Non-Uniform Data

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": {
         "date": 1997,
         "product": "Furby"
      }
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- What if data is not uniform?

Unnesting Non-Uniform Data

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": {
         "date": 1997,
         "product": "Furby"
      }
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- What if data is not uniform?

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p UNNEST p.orders AS o;
```

Why is this now invalid?

Unnesting Non-Uniform Data

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": {
         "date": 1997,
         "product": "Furby"
      }
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- What if data is not uniform?

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p UNNEST p.orders AS o;
```

Why is this now invalid?

Can't query on an object! Only arrays and multisets

Unnesting Non-Uniform Data

```
-- Dataset Person
{{
   {
      "name": "Dan",
      "phone": "555-123-4567",
      "orders": {
         "date": 1997,
         "product": "Furby"
      }
   },
   {
      "name": "Alvin",
      "phone": "555-234-5678",
      "orders": [
         {
            "date": 2000,
            "product": "Furby"
         },
         {
            "date": 2012,
            "product": "Magic8"
         }
      ]
   },
   {
      "name": "Magda",
      "phone": "555-345-6789",
      "orders": []
   }
}}
```

- What if data is not uniform?
  - Use built-in functions/keywords

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p,
       (CASE WHEN p.orders IS MISSING
             THEN []
             WHEN IS_ARRAY(p.orders)
             THEN p.orders
             ELSE [p.orders]
             END
       ) AS o;
```

Unnesting Non-Uniform Data

- Useful functions
  - IS_ARRAY(…)
  - IS_OBJECT(…)
  - IS_BOOLEAN(…)
  - IS_STRING(…)
  - IS_NUMBER(…)
  - IS_NULL(…)
  - IS_MISSING(…)
  - IS_UNKNOWN(…)

Unnested Data → Nested Results

- Long story short:
  - Correlated SELECT subquery
  - From the documentation: "Note that a subquery, like a top-level SELECT statement, **always returns a collection** – regardless of where within a query the subquery occurs."

Unnested Data → Nested Results

Different query!

Return a object for each product and a list of people who bought that product.

```
-- Dataset Orders
{{
   {
      "pname": "Dan"
      "date": 1997,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2000,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2012,
      "product": "Magic8"
   }
}}
```

Unnested Data → Nested Results

Different query!

Return a object for each product and a list of people who bought that product.

```
SELECT DISTINCT o.product,
       (SELECT u.pname
          FROM Orders AS u
         WHERE o.product = u.product) AS names
  FROM Orders AS o;
```

Note this would error in SQL!

```
-- Dataset Orders
{{
   {
      "pname": "Dan"
      "date": 1997,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2000,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2012,
      "product": "Magic8"
   }
}}
```

Unnested Data → Nested Results

Different query!

Return a object for each product and a list of people who bought that product.

```
SELECT DISTINCT o.product, n AS names
  FROM Orders AS o
   LET n = (SELECT u.pname
              FROM Orders AS u
             WHERE o.product = u.product);
```

Cleaner (I think)

```
-- Dataset Orders
{{
   {
      "pname": "Dan"
      "date": 1997,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2000,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2012,
      "product": "Magic8"
   }
}}
```

Unnested Data → Nested Results

Different query!

Return a object for each product and a list of people who bought that product.

```
-- Dataset Orders
{{
   {
      "pname": "Dan"
      "date": 1997,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2000,
      "product": "Furby"
   },
   {
      "pname": "Alvin"
      "date": 2012,
      "product": "Magic8"
   }
}}
```

```
SELECT DISTINCT o.product, n AS names
  FROM Orders AS o
   LET n = (SELECT u.pname
              FROM Orders AS u
             WHERE o.product = u.product);
```

```
-- Output
/*
{product: Furby, names:[{pname: Dan}, {pname: Alvin}]}
{product: Magic8, names:[{pname: Alvin}]}
*/
```

Takeaways

- Semi-structured data is best for **data exchange**
- Best practices
  - Use SQL++ and other semi-structured native query languages for ad-hoc analysis
    - Ever tried doing ctrl-f on JSON data?
  - Pay attention to human side of things!
    - Most advanced engines like AsterixDB can "run as fast" as a RDBMS
    - Like all things in CS, make sure others can understand it!
    - **Long-term data analysis will benefit from time spent up front to normalize data into a RDBMS**