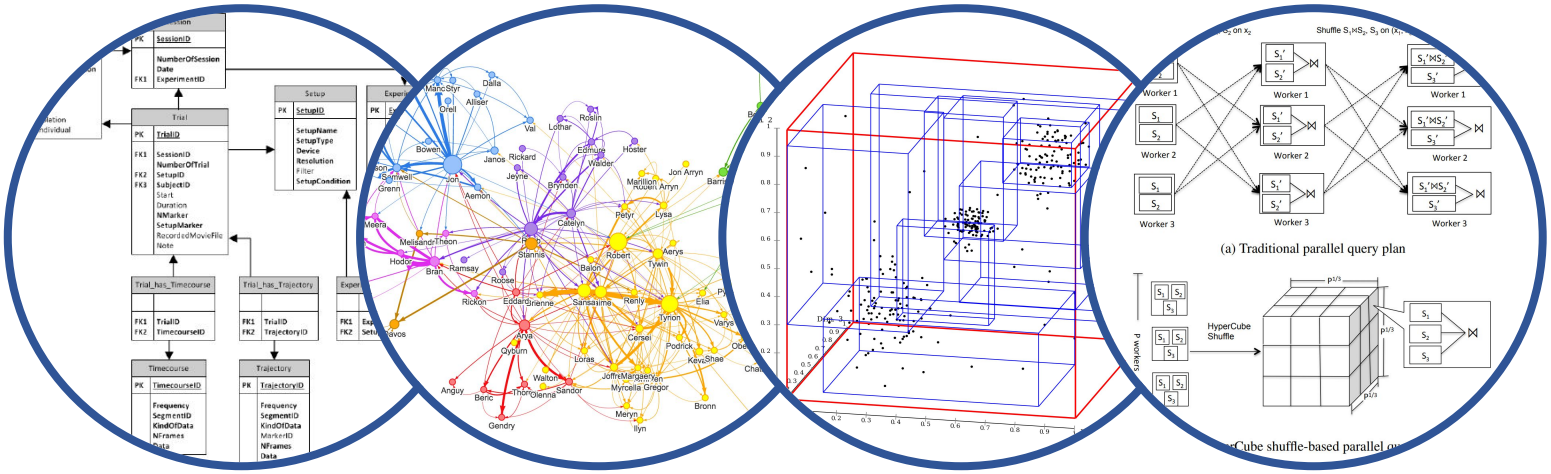


Introduction to Data Management

SQL++

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

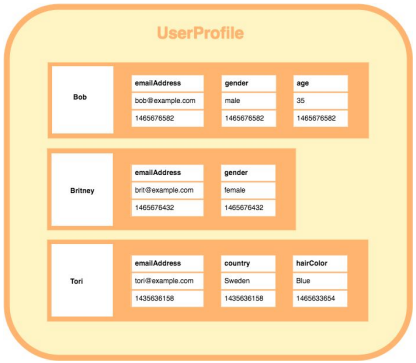
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Recap: NoSQL Data Models

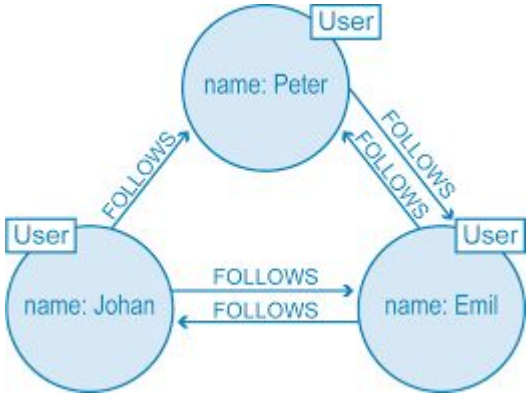
Key-Value Database

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Wide-Column Store (Extensible Record Store)



Graph Database



Document Store

XML

```
<empinfo>
   <employees>
      <employee>
         <name>James Kirk</name>
         <age>40</age>
      </employee>
      <employee>
         <name>Jean-Luc Picard</name>
         <age>45</age>
      </employee>
      <employee>
         <name>Wesley Crusher</name>
         <age>27</age>
      </employee>
   </employees>
</empinfo>
```

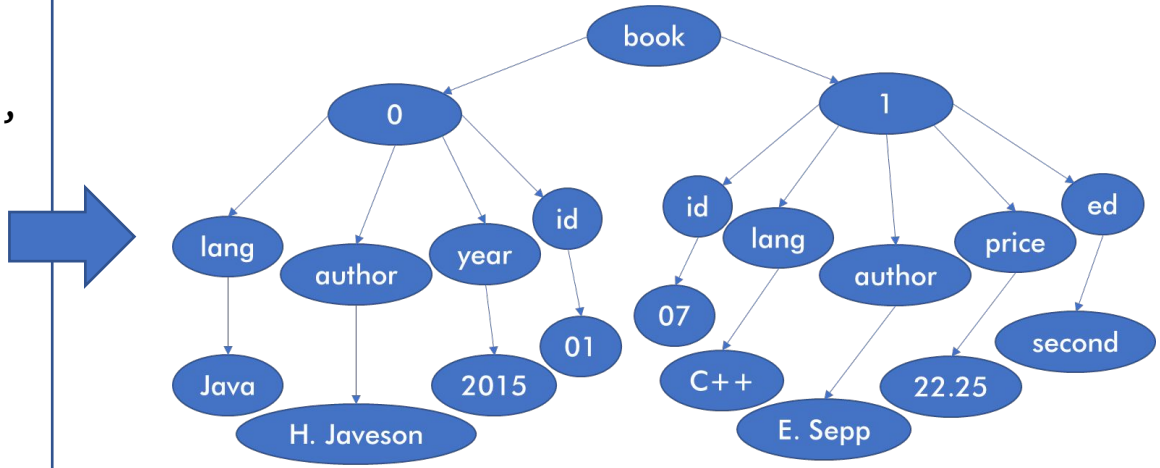
JSON

```
{  "empinfo" :
   {
      "employees" :  [
      {
         "name" : "James Kirk",
         "age" : 40,
      },
      {
         "name" : "Jean-Luc Picard",
         "age" : 45,
      },
      {
         "name" : "Wesley Crusher",
         "age" : 27,
      }
                  ]
   }
}
```

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: Tradeoffs of Semi-Structured Data

Pros	Cons
More **flexible** data (not restricted to first normal form)	Data can become arbitrary and **hard to reason** about Uniform objects can be extremely **redundant** with the embedded schema
Easy **data exchange** due to schema being baked in	Requires **parsing** (rather than direct access/search) to get data
We can **"precompute" joins** that can lead to speedups	Nesting data makes **"complex" queries harder**

Today

- AsterixDB as a case study of Document Store
  - Semi-structured data model in JSON
  - **Introducing AsterixDB and SQL++**

Today

Today:

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

The 5 W's of AsterixDB

- Who
  - M. J. Carey & co.
- What
  - "A Scalable, Open Source BDMS"
  - It is now also an Apache project
- Where
  - UC Irvine, Cloudera Inc, Google, IBM, ...
- When
  - 2014
- Why
  - To develop a next-gen system for managing semi-structured data

The 5 W's of SQL++

- Who
  - K. W. Ong & Y. Papakonstantinou
- What
  - A query language that is applicable to JSON native stores and SQL databases
- Where
  - UC San Diego
- When
  - 2015
- Why
  - Stand in for other semi-structured query languages that lack formal semantics.

Why We are Choosing SQL++

- Strong formal semantics
  - Original paper: https://arxiv.org/pdf/1405.3631.pdf
  - Nested relational algebra: https://dl.acm.org/citation.cfm?id=588133
- Many systems adopting or converging to SQL++
  - Apache AsterixDB
  - CouchBase (N1QL)
  - Apache Drill
  - Snowflake

Asterix Data Model (ADM)

- Nearly identical to the JSON standard
- Some additions
  - New primitive: **universally unique identifier (uuid)**
    - Ex: 123e4567-e89b-12d3-a456-426655440000
  - New derived type: **multiset**
    - A bag - unordered collection permitting duplicates
    - Encapsulated by double curly braces {{ }}
- Queried data must be a multiset or array

Introducing the New and Improved SQL++

Today

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?

- You have seen it all before!

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

Schema Manipulation

DDL? DML?

- You have seen it all before!

	SQL Examples	SQL++ Examples
Data Description Language (DDL)	C C C D A	C
Data Manipulation Language (DML)	S I D	S

Schema Manipulation

Data Manipulation

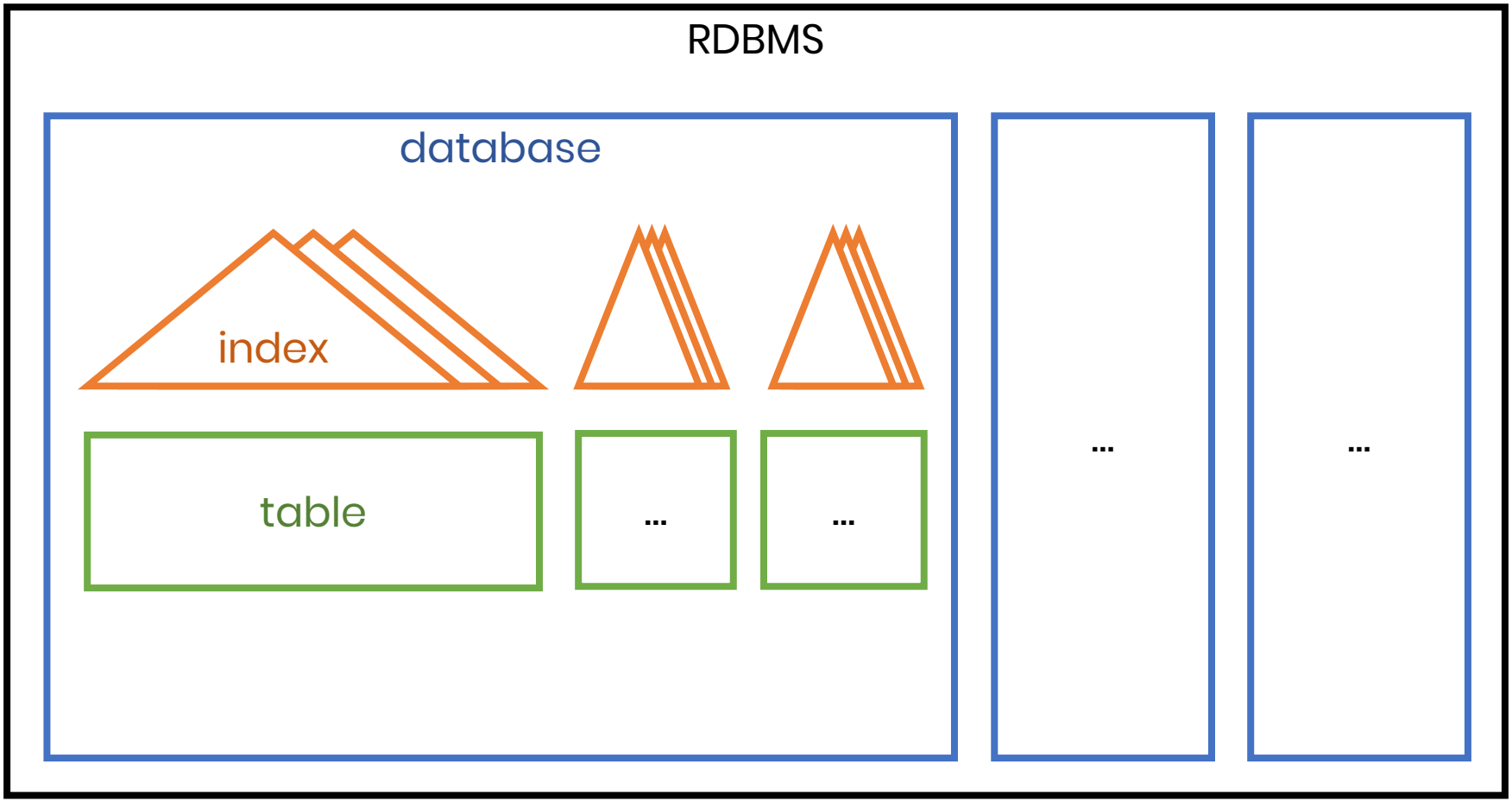
Data Definition Language (DDL)

- 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**

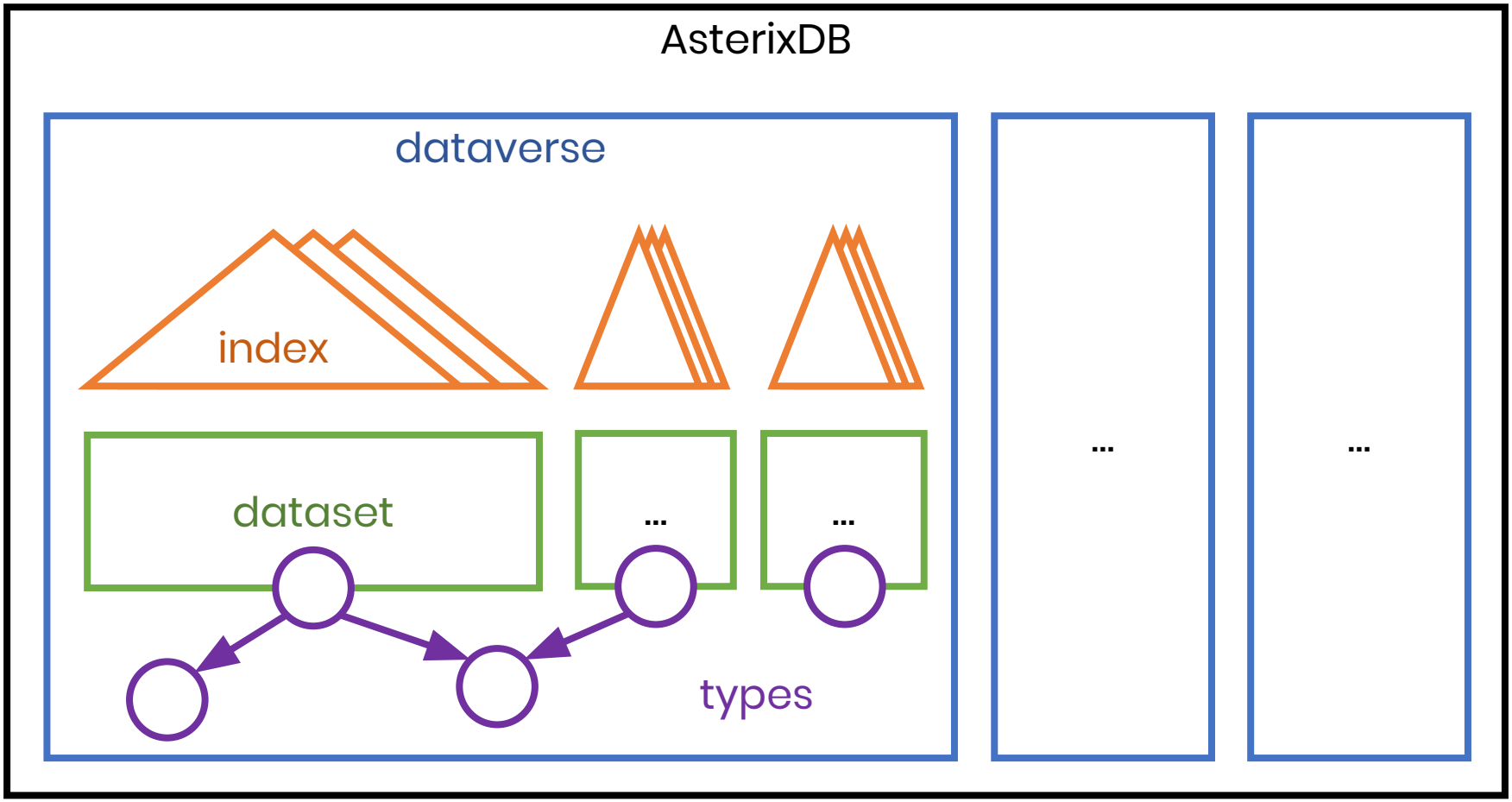
Data Definition Language (DDL)

- Extremely similar to the relational world



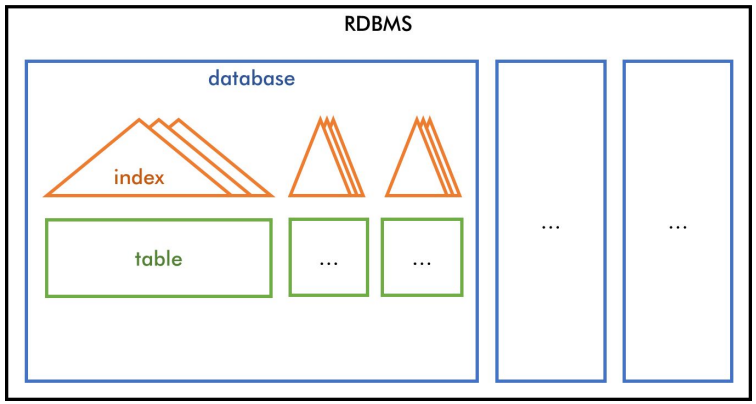
Data Definition Language (DDL)

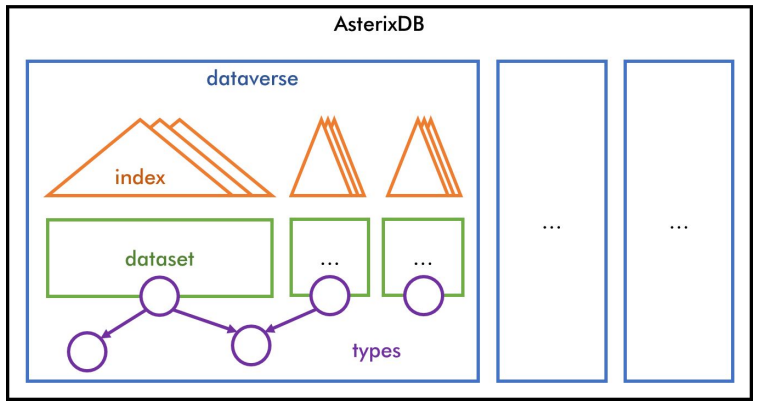
- Extremely similar to the relational world



Data Definition Language (DDL)

- Extremely similar to the relational world





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

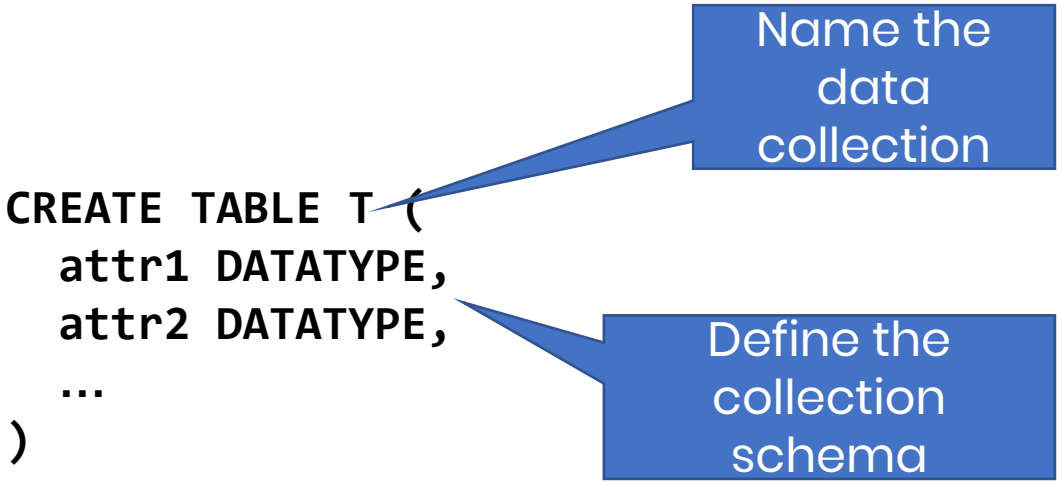
Types

What is this SQL statement doing?

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

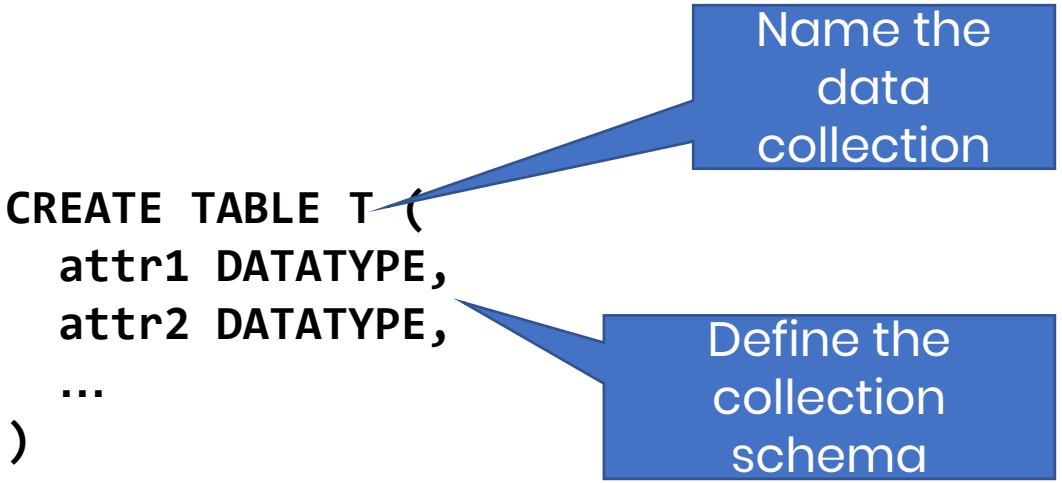
Types

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

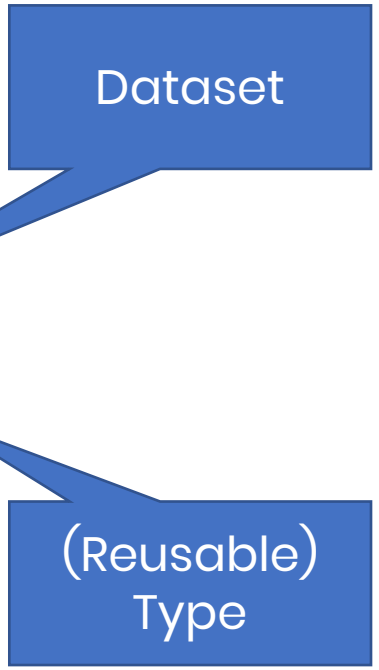
Less abstraction!

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

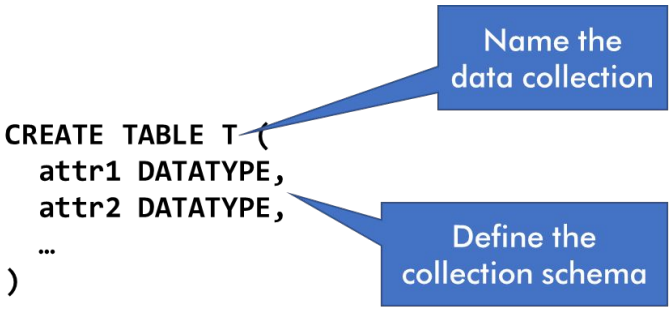
Types

Less abstraction!

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



Types



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 (by default)

Types

- Ensures adherence to schema

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

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

Types

- Ensures adherence to schema

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

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

Types

- Ensures adherence to schema

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

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

Types

- Ensures adherence to schema

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

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

Can't be missing required fields

Types

- Ensures adherence to schema

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

All the checks we've seen so far apply to both CLOSED and OPEN types

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

Can't be missing required fields

Open Types

- Allows additional fields

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

OPEN by default

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

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

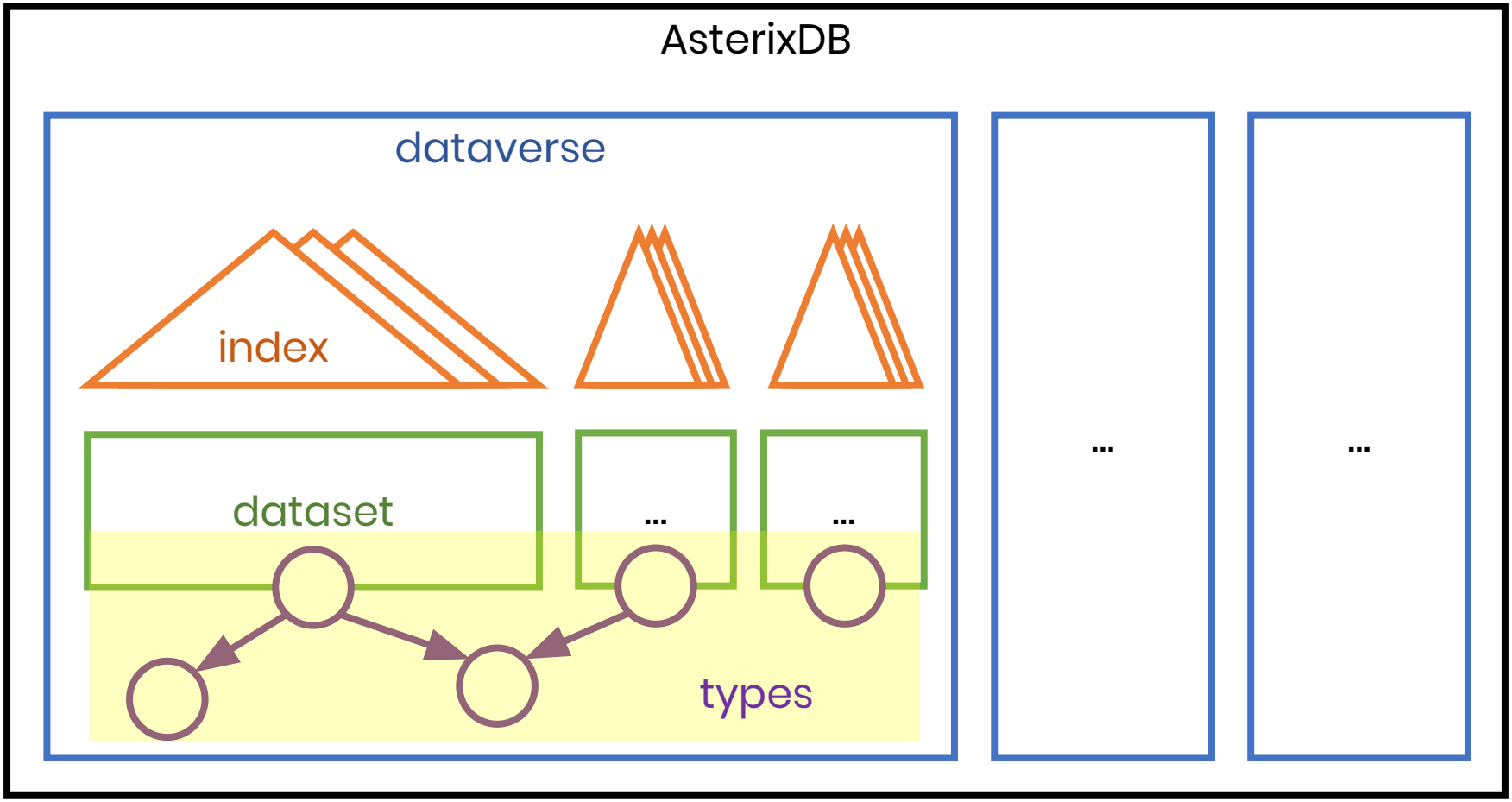
Collection Data Types

- Datatype can be a collection

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

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

Types Within Types



Types Within Types

- Tree structure!

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

```
USE myDB;
DROP TYPE ContactType IF EXISTS;
CREATE TYPE ContactType AS CLOSED {
  method:  string,
  contactStr: string
}
```

```
[
   {
      "name": "Dan",
      "contact": {
         "method": "phone",
         "contactStr": "5551234567"
      }
   }
]
```

Types Within Types

- Tree structure!

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

```
USE myDB;
DROP TYPE ContactType IF EXISTS;
CREATE TYPE ContactType AS CLOSED {
  method:  string,
  contactStr: string
}
```

```
[
   {
      "name": "Dan",
      "contact": [
         {
            "method": "phone",
            "contactStr": "5551234567"
         },
         {
            "method": "email",
            "contactStr": "suciu@cs"
         }
      ]
   }
]
```

Types Within Types

- Tree structure!

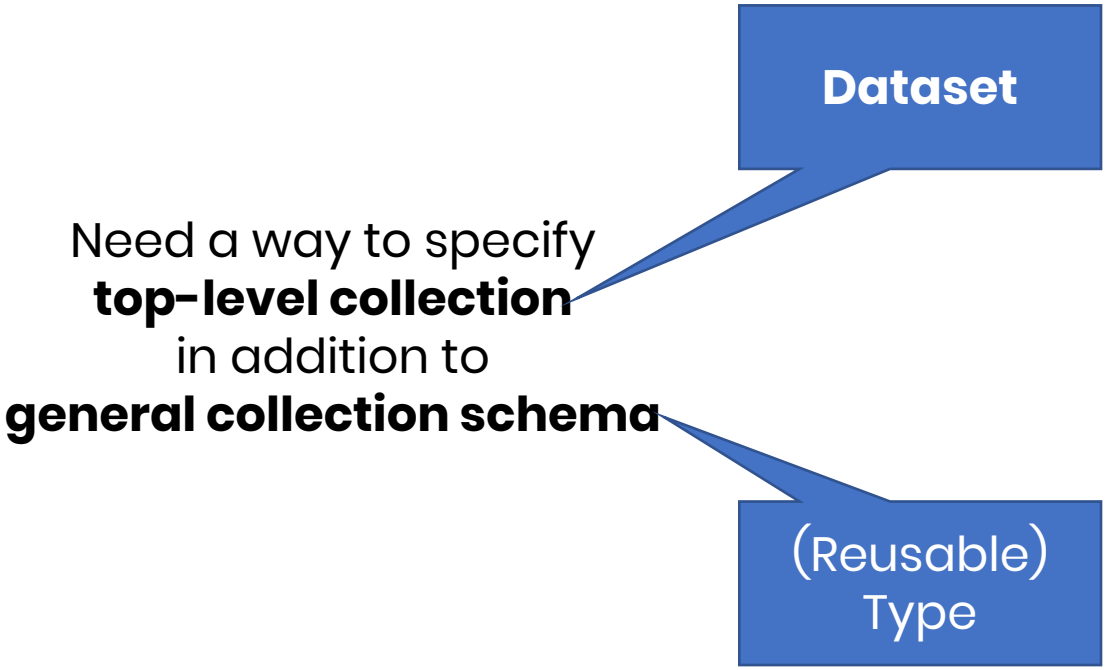
```
USE myDB;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE Perso Type AS CLOSED
  name:  string,
  contact: [Contac
}
```

```
USE myDB;
DROP TYP actType IF EXISTS;
ntactTyp AS CLOSED {
```

Goodbye,
first normal form!

```
5512345678
        {
            "method": "email",
            "contactStr": "suciu@cs"
        }
    ]
  }
]
```

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?

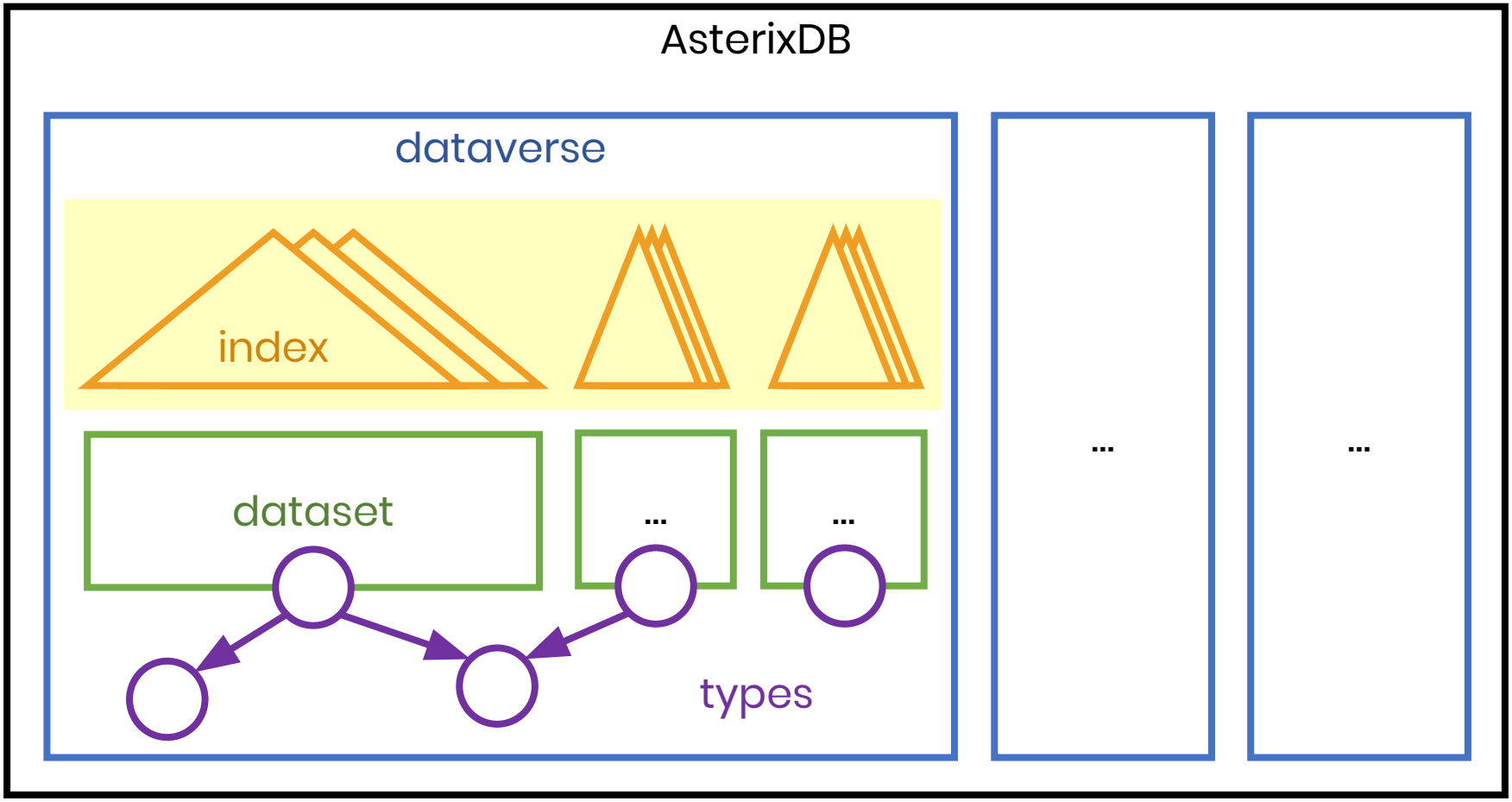
Autogenerate!

```
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"

Indexing

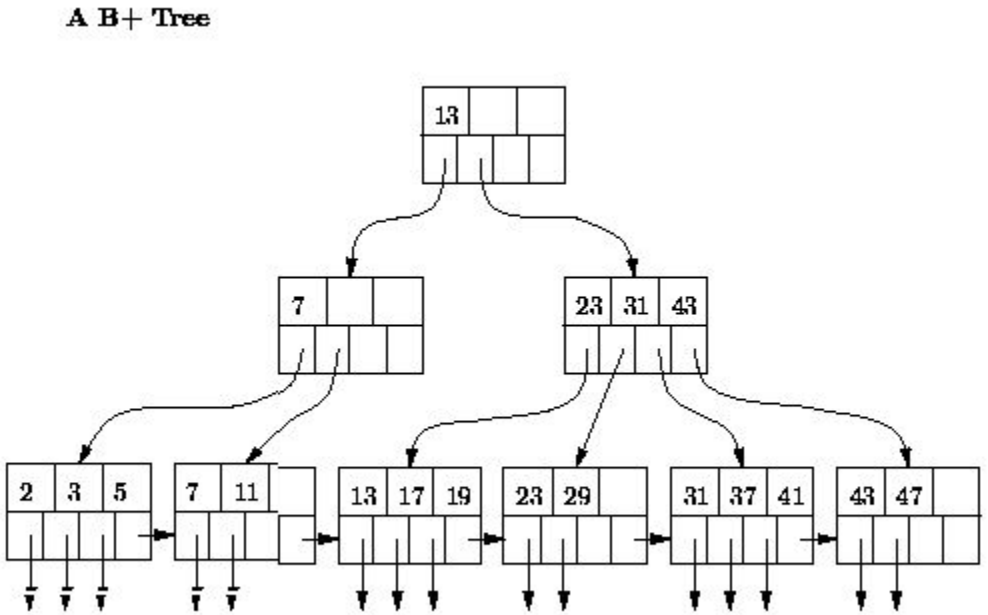


Index Zoo

- BTREE
- RTREE
- KEYWORD
- NGRAM

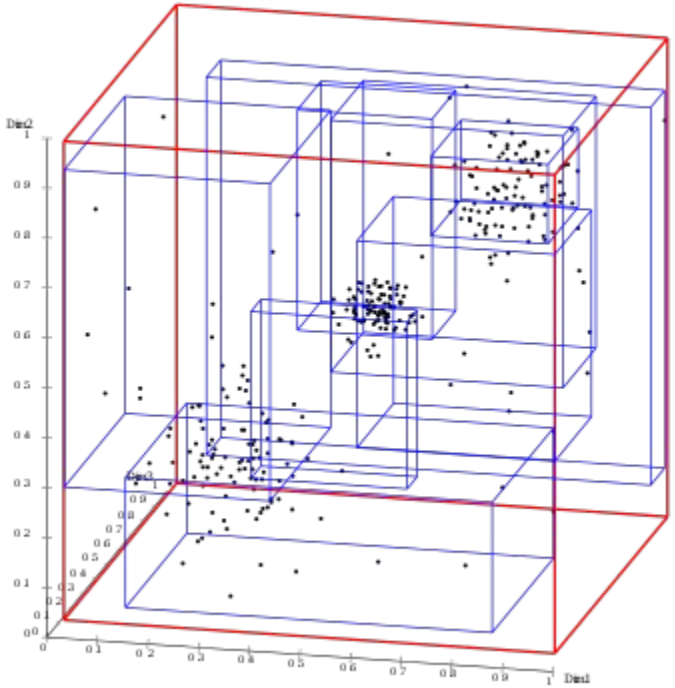
Index Zoo

- **BTREE**
- RTREE
- KEYWORD
- NGRAM



Index Zoo

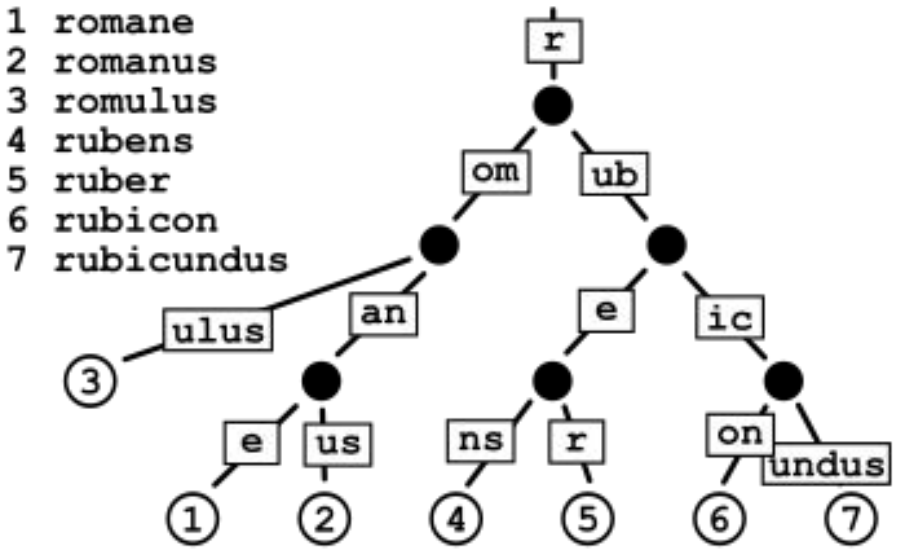
- BTREE
- **RTREE**
- KEYWORD
- NGRAM



Multi-dimensional B-Tree

Index Zoo

- BTREE
- RTREE
- **KEYWORD**
- NGRAM



Radix tree

Index Zoo

- Can only index on top-level fields, not nested fields

```
USE myDB;
CREATE INDEX ContactName ON
Person(name) TYPE BTREE;
```

~~`USE myDB; CREATE INDEX ContactName ON Person(contact.method) TYPE BTREE;`~~

```
[
   {
      "name": "Dan",
      "contact": [
         {
            "method": "phone",
            "contactStr": "5551234567"
         },
         {
            "method": "email",
            "contactStr": "suciu@cs"
         }
      ]
   }
]
```

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;
```

```
-- 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"
   }
}}
```

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"
   }
}}
```

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;

-- 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}
*/
```

```
-- 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"
   }
}}
```

Joins

- Omits fields for OUTER JOIN no-match

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p LEFT OUTER JOIN
       Orders AS o
    ON p.name = o.pname;
```

```
-- 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"
   }
}}
```

Joins

- Omits fields for OUTER JOIN no-match

```
SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p LEFT OUTER JOIN
       Orders AS o
    ON p.name = o.pname;

-- 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}
{name: Magda, phone: 555-345-6789}
*/
```

```
-- 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"
   }
}}
```

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)
  - Similar process to flatmap

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)
  - Similar process to flatmap

```
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)
  - Similar process to flatmap

```
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)
  - Similar process to flatmap

```
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}
*/
```

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)
  - Similar process to flatmap

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

-- output
/*
{name: Dan,   phone:                    oduct: Furby}
{name: Alvin, phone:                    oduct: Furby}
{name: Alvin, phone:                    oduct: Magic8}
*/
```

Implicitly knows to UNNEST since we reference the other dataset

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)
  - Similar process to flatmap

```
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: 555-234-567 , 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": []
   }
}}
```

- 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"
   }
}}
```

- 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"
   }
}}
```

- 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"
   }
}}
```

- 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! Or a missing field!

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"
   }
}}
```

- What if data is not uniform?
  - Use built-in functions/keywords to let the query deal with it uniformly

```
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]
       ) 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);
```

Alternate syntax

```
-- 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**