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
Thanksgiving holiday this week
  • No section Thursday
  • No class Friday
  • No Th/F office hours
    • (I’ll have mine W but may leave early if no one is there)
  • Minimal message board coverage for holidays
Recap: NoSQL Data Models

**Key-Value Database**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>AAA, BBB, CCC</td>
</tr>
<tr>
<td>K2</td>
<td>AAA, BBB</td>
</tr>
<tr>
<td>K3</td>
<td>AAA, DDD</td>
</tr>
<tr>
<td>K4</td>
<td>AAA, 2,01/01/2015</td>
</tr>
<tr>
<td>K5</td>
<td>3, ZZZ, 5623</td>
</tr>
</tbody>
</table>

**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>
```

**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

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>More <strong>flexible</strong> data (not restricted to first normal form)</td>
<td>Data can become arbitrary and <strong>hard to reason</strong> about</td>
</tr>
<tr>
<td></td>
<td>Uniform objects can be extremely <strong>redundant</strong> with the embedded schema</td>
</tr>
<tr>
<td>Easy <strong>data exchange</strong> due to schema being baked in</td>
<td>Requires <strong>parsing</strong> (rather than direct access/search) to get data</td>
</tr>
<tr>
<td>We can &quot;<strong>precompute</strong>&quot; joins that can lead to speedups</td>
<td>Nesting data makes &quot;<strong>complex</strong>&quot; queries harder</td>
</tr>
</tbody>
</table>
Today

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

- 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!

<table>
<thead>
<tr>
<th>Data Description Language (DDL)</th>
<th>SQL Examples</th>
<th>SQL++ Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CREATE DATABASE…</td>
<td>CREATE DATaverse…</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE…</td>
<td>CREATE TYPE… (unique)</td>
</tr>
<tr>
<td></td>
<td>CREATE INDEX…</td>
<td>CREATE DATASET…</td>
</tr>
<tr>
<td></td>
<td>DROP TABLE… (unique)</td>
<td>CREATE INDEX…</td>
</tr>
<tr>
<td></td>
<td>ALTER TABLE…</td>
<td>DROP DATASET…</td>
</tr>
<tr>
<td>Data Manipulation Language (DML)</td>
<td>SELECT…FROM…</td>
<td>SELECT…FROM…</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO…</td>
<td>INSERT INTO…</td>
</tr>
<tr>
<td></td>
<td>DELETE FROM…</td>
<td>DELETE FROM…</td>
</tr>
</tbody>
</table>
### DDL? DML?

- You have seen it all before!

<table>
<thead>
<tr>
<th>Data Description Language (DDL)</th>
<th>SQL Examples</th>
<th>SQL++ Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE DATABASE…</td>
<td>CREATE DATABASE…</td>
<td>CREATE DATAVERSE…</td>
</tr>
<tr>
<td>CREATE TABLE…</td>
<td>CREATE TABLE…</td>
<td>CREATE DATASET…</td>
</tr>
<tr>
<td>CREATE INDEX…</td>
<td>CREATE INDEX…</td>
<td>CREATE INDEX…</td>
</tr>
<tr>
<td>DROP TABLE…</td>
<td>DROP TABLE…</td>
<td>DROP DATASET…</td>
</tr>
<tr>
<td>ALTER TABLE… (unique)</td>
<td>ALTER TABLE…</td>
<td>ALTER TABLE…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Manipulation Language (DML)</th>
<th>SQL Examples</th>
<th>SQL++ Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT…FROM…</td>
<td>SELECT…FROM…</td>
<td>SELECT…FROM…</td>
</tr>
<tr>
<td>INSERT INTO…</td>
<td>INSERT INTO…</td>
<td>INSERT INTO…</td>
</tr>
<tr>
<td>DELETE FROM…</td>
<td>DELETE FROM…</td>
<td>DELETE FROM…</td>
</tr>
</tbody>
</table>

**Schema Manipulation**
### DDL? DML?

- You have seen it all before!

<table>
<thead>
<tr>
<th></th>
<th>SQL Examples</th>
<th>SQL++ Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Description Language (DDL)</strong></td>
<td>CREATE DATABASE…</td>
<td>CREATE DATaverse…</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE…</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CREATE INDEX…</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DROP TABLE…</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALTER TABLE… (unique)</td>
<td></td>
</tr>
<tr>
<td><strong>Data Manipulation Language (DML)</strong></td>
<td>SELECT… FROM…</td>
<td>SELECT… FROM…</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO…</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DELETE FROM…</td>
<td></td>
</tr>
</tbody>
</table>

- **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

AsterixDB

- dataverse
- index
- dataset
- types
Data Definition Language (DDL)

- Extremely similar to the relational world

<table>
<thead>
<tr>
<th>Functionality</th>
<th>RDBMS</th>
<th>AsterixDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace</td>
<td>Database</td>
<td>Dataverse</td>
</tr>
<tr>
<td>Data Collection</td>
<td>Table</td>
<td>Dataset</td>
</tr>
<tr>
<td>Data Access</td>
<td>Index</td>
<td>Index</td>
</tr>
</tbody>
</table>
What is this SQL statement doing?

```
CREATE TABLE T (
    attr1 DATATYPE,
    attr2 DATATYPE,
    ...
)
```
What is this SQL statement doing?

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

Name the data collection
Define the collection schema
What is this SQL statement doing?

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

Flat data can do it all in one step!
What about nested data?
```json
{
  "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

```json
{
  "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": []
    }
  ]
}
```

Need to describe person schema
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": []  
    }  
  ]  
}
```

Need to describe person schema

Person schema needs orders 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

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 (by default)
Ensures adherence to schema

```sql
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"
    }
]
```
- Ensures adherence to schema

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

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

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

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

Can't be missing required fields

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

- Allows additional fields

```sql
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,
        "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)

```sql
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
**Datatype can be a collection**

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

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

AsterixDB

dataverse

index

dataset

types

...
Types Within Types

- Tree structure!

```sql
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
}
```

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

- Tree structure!

```sql
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 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"
            }
        ]
    }
]
Datasets

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

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

```sql
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

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;

What if there are no good keys?
Dataset Keys

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

What if there are no good keys?

Autogenerate!

```sql
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

```
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;
```

What if there are no good keys?

Autogenerate!

Each object will have a uuid field named "myKey"
Indexing

AsterixDB

dataverse

index

dataset

types

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

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

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

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

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

- Same nested-loop semantics as SQL!

```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:
      ...

```json
-- 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!

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

**-- Output**

```sql
/*
{name: "Dan", phone: 555-123-4567, date: 1997, product: "Furby"}
{name: "Alvin", phone: 555-234-5678, date: 2000, product: "Furby"}
*/
```
Joins

- Omits fields for OUTER JOIN no-match

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

```json
-- 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

```sql
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}
*/
Nested Data

- Two interesting directions
  - Nested data ➔ Unnested results
  - Unnested data ➔ Nested results
Nested Data  Unnested Results

- How do we **unnest** data?

```sql
-- 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": []}
}}
```
Nested Data ▶ Unnested Results

How do we **unnest** data?

- SQL++ can unnest and join all at once (built into syntax)
- Similar process to flatmap

---

```
-- 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": []
}
```
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

---

**How do we **unnest** data?**

- SQL++ can unnest and join all at once (built into syntax)
- Similar process to flatmap

```sql
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 \(\text{→}\) Unnested Results

- **Dataset Person**

```json
{{
  {
    "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

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

-- output

```javascript
/*
{name: "Dan", phone: 555-123-4567, date: 1997, product: "Furby"}
{name: "Alvin", phone: 555-234-5678, date: 2000, product: "Furby"}
*/
```
Nested Data  Unnested Results

--- Dataset Person

```json
{{
  "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

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

---

Implicitly knows to UNNEST since we reference the other dataset

```*/
{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, orders: []}
*/*
```
### Nested Data ▸ Unnested Results

---

**Dataset Person**

```json
{{
    "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

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

---

**Output**

```sql
/*
{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

- What if data is not uniform?

```sql
-- 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": []
  }
}
```
Unnesting Non-Uniform Data

- What if data is not uniform?

```sql
-- 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"
}
}
```
Unnesting Non-Uniform Data

- What if data is not uniform?

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

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

Why is this now invalid?
What if data is not uniform?

```sql
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

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

```sql
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(...)`

Is value NULL or MISSING?
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."
Different query!
Return a object for each product and a list of people who bought that product.

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

-- Dataset Orders
{{
    {
        "pname": "Dan",
        "date": 1997,
        "product": "Furby"
    },
    {
        "pname": "Alvin",
        "date": 2000,
        "product": "Furby"
    },
    {
        "pname": "Alvin",
        "date": 2012,
        "product": "Magic8"
    }
}}
Different query!
Return a object for each product and a list of people who bought that product.

```sql
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

```json
/*
{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**