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

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

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?

<table>
<thead>
<tr>
<th>SQL Examples</th>
<th>SQL++ Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE DATABASE...</td>
<td>CREATE DATASVERSE...</td>
</tr>
<tr>
<td>CREATE TABLE...</td>
<td>CREATE TYPE... (unique)</td>
</tr>
<tr>
<td>CREATE INDEX...</td>
<td>CREATE DATASET...</td>
</tr>
<tr>
<td>DROP TABLE...</td>
<td>CREATE INDEX...</td>
</tr>
<tr>
<td>ALTER TABLE...</td>
<td>DROP DATASET...</td>
</tr>
<tr>
<td>SELECT... FROM...</td>
<td>SELECT... FROM...</td>
</tr>
<tr>
<td>INSERT INTO...</td>
<td>INSERT INTO...</td>
</tr>
<tr>
<td>DELETE FROM...</td>
<td>DELETE FROM...</td>
</tr>
</tbody>
</table>
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>SELECT FROM...</td>
<td>CREATE DATABASE</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO...</td>
<td>CREATE TABLE...</td>
</tr>
<tr>
<td></td>
<td>DELETE FROM...</td>
<td>CREATE INDEX...</td>
</tr>
<tr>
<td></td>
<td>CREATE DATABASE</td>
<td>DROP DATABASE...</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE...</td>
<td>CREATE TABLE...</td>
</tr>
<tr>
<td></td>
<td>CREATE INDEX...</td>
<td>DROP INDEX...</td>
</tr>
</tbody>
</table>

Data Manipulation Language (DML)

<table>
<thead>
<tr>
<th>Data Manipulation Language (DML)</th>
<th>SQL Examples</th>
<th>SQL++ Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT FROM...</td>
<td>CREATE DATABASE</td>
</tr>
<tr>
<td></td>
<td>INSERT INTO...</td>
<td>CREATE TABLE...</td>
</tr>
<tr>
<td></td>
<td>DELETE FROM...</td>
<td>CREATE INDEX...</td>
</tr>
<tr>
<td></td>
<td>CREATE DATABASE</td>
<td>DROP DATABASE...</td>
</tr>
<tr>
<td></td>
<td>CREATE TABLE...</td>
<td>CREATE TABLE...</td>
</tr>
<tr>
<td></td>
<td>CREATE INDEX...</td>
<td>DROP INDEX...</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Functionality</th>
<th>DBMS</th>
<th>AsterixDB</th>
</tr>
</thead>
<tbody>
<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>

Types

What is this SQL statement doing?

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

Define the collection schema

What about nested data?

```
{ 
  "person": [ 
    { 
      "name": "Dan", 
      "phone": "555-123-4567", 
      "orders": [ 
        { 
          "date": 1997, 
          "product": "Furby"
        },
        { 
          "date": 2012, 
          "product": "Magic8"
        }
      ]
    },
    { 
      "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 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)

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

Can't be missing required fields

Open Types

- Allows additional fields

```sql
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
  }
]
```

Can't use unspecified fields
Collection Data Types

- Data can be a collection

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

  Mean phone is an array of ints

Datasets

- 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;
  ```

  Each object will have a uuid field named "myKey"

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 myKey AUTOGENERATED;
  ```

  [Reusable Type]

  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;
  ```

  What if there are no good keys?

  ```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;
  ```

  Autogenerate!
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;
```

**Nested Data**

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

**SQL++ Aggregation**

Better encapsulation of 3-valued logic!

- NULL considered
- NULL ignored (same as vanilla SQL)

**Nested Data → Unnested Results**

- How do we unnest data?
Nested Data → Unnest Results

- 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

• Derreferencing can only be done on objects!

• 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, p.orders AS o;
-- output
/*
*/

• 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
/*
*/

• 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, p.orders AS o;
-- error
### Nested Data → Unnesting Results

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

  ```sql
  SELECT p.name, p.phone, o.date, o.product
  FROM Person AS p, UNNEST p.orders AS o;
  -- output
<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alvin</td>
<td>555-345</td>
<td>2012</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-345</td>
<td>2012</td>
<td>Magic8</td>
</tr>
<tr>
<td>Dan</td>
<td>555-123</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Dan</td>
<td>555-123</td>
<td>2000</td>
<td>Magic8</td>
</tr>
<tr>
<td>Magda</td>
<td>678-999</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Magda</td>
<td>678-999</td>
<td>1997</td>
<td>Magic8</td>
</tr>
</tbody>
</table>
  ```

- **Parent-child join!**

### Unnesting Non-Uniform Data

- **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;
  -- output
<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alvin</td>
<td>555-345</td>
<td>2012</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>555-345</td>
<td>2012</td>
<td>Magic8</td>
</tr>
<tr>
<td>Dan</td>
<td>555-123</td>
<td>2000</td>
<td>Furby</td>
</tr>
<tr>
<td>Dan</td>
<td>555-123</td>
<td>2000</td>
<td>Magic8</td>
</tr>
<tr>
<td>Magda</td>
<td>678-999</td>
<td>1997</td>
<td>Furby</td>
</tr>
<tr>
<td>Magda</td>
<td>678-999</td>
<td>1997</td>
<td>Magic8</td>
</tr>
</tbody>
</table>
  ```

- **Why is this now invalid?**

  - Can’t query on an object
  - Only arrays and multisets

- **Use built-in functions/keywords**

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

Unnesting 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.

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);

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);

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);

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);
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