ADMINISTRATIVE MINUTIAE

• Exam Friday
• Covers material through today’s lecture
• HW4 due Friday, 11:30 pm
• OQ4 due Wednesday, 11:00 pm
  • HW5, OQ5 out next week
EXAM MINUTIAE

• No note sheets or resources
• 1:20 to complete
• 7-10 questions
  • Short answer
  • SQL code
  • Relational Algebra
  • Datalog
EXAM MINUTIAE

• SQL and Datalog code fair game for exams
  • Watch for syntax
• No HW assignment on SQL++ before midterm
  • Only need to know applications and differences, no SQL++ code on midterm
  • Dataverse, type, dataset, index
EXAM MINUTIAE

• Database design principles
  • Good design decisions
  • New properties
  • Keys and indices

• Previous exams
  • Good resource for studying

• Extra office hours: 12:00-2:00 Wednesday
  CSE 214
EXAM MINUTIAE

• Practice exam
  • Out tomorrow
  • Solutions in section on Thursday

• Exam review
  • In class Wednesday
  • Topics and likely questions
WHERE WE ARE

So far we have studied the **relational data model**

- Data is stored in tables (=relations)
- Queries are expressions in SQL, relational algebra, or Datalog

This week: **Semistructured data model**

- Popular formats today: XML, JSon, protobuf
DATA MODELS

Taxonomy based on data models:

**Key-value stores**
- e.g., Project Voldemort, Memcached

**Document stores**
- e.g., SimpleDB, CouchDB, MongoDB

**Extensible Record Stores**
- e.g., HBase, Cassandra, PNUTS
MOTIVATION

In Key, Value stores, the Value is often a very complex object

- Key = ‘2010/7/1’, Value = [all flights that date]

Better: allow DBMS to understand the value

- Represent value as a JSON (or XML...) document
- [all flights on that date] = a JSON file
- May search for all flights on a given date
ASTERIXDB AND SQL++

AsterixDB
- No-SQL database system
- Developed at UC Irvine
- Now an Apache project
- Own query language: AsterixQL or AQL, based on XQuery

SQL++
- SQL-like syntax for AsterixQL
EXAMPLES

Try these queries:

```sql
SELECT x.age FROM [{'name': 'Alice', 'age': ['30', '50']}] x;
```

```sql
SELECT x.age FROM {{ {'name': 'Alice', 'age': ['30', '50']}}} x;
```

```sql
-- error
SELECT x.age FROM {'name': 'Alice', 'age': ['30', '50']} x;
```

Can only select from multi-set or array
DATATYPES

Boolean, integer, float (various precisions), geometry (point, line, …), date, time, etc

UUID = universally unique identifier
Use it as a system-generated unique key
SQL++ OVERVIEW

Data Definition Language (DDL): create a
  • Dataverse
  • Type
  • Dataset
  • Index

Data Manipulation Language (DML): select-from-where
A Dataverse is a Database

CREATE DATaverse lec344

CREATE DATaverse lec344 IF NOT EXISTS

DROP DATaverse lec344

DROP DATaverse lec344 IF EXISTS

USE lec344
TYPE

Defines the schema of a collection
It lists all **required** fields
Fields followed by ? are **optional**
CLOSED type = no other fields allowed
OPEN type = other fields allowed
CLOSED TYPES

USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    Name : string,
    age: int,
    email: string?
}

{"Name": "Alice", "age": 30, "email": "a@alice.com"}

{"Name": "Bob", "age": 40}

-- not OK:
{"Name": "Carol", "phone": "123456789"}
**OPEN TYPES**

```sql
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
    Name : string,
    age: int,
    email: string?
}

{"Name": "Alice", "age": 30, "email": "a@alice.com"}

{"Name": "Bob", "age": 40}

-- Now it’s OK:
{"Name": "Carol", “age”: 50, "phone": "123456789"}
```
TYPES WITH NESTED COLLECTIONS

USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    Name : string,
    phone: [string]
}

{"Name": "Carol", "phone": ["1234"]}
{"Name": "David", "phone": ["2345", "6789"]}
{"Name": "Eric", "phone": []}
DATASETS

Dataset = relation

Must have a type
  • Can be a trivial OPEN type

Must have a key
  • Can also be a trivial one
DATASET WITH AUTOGENERATED KEY

USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
    myKey: uuid,
    Name : string,
    email: string?
}

USE lec344;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType)
    PRIMARY KEY myKey AUTOGENERATED;

Note: no myKey since it will be autogenerated

{“Name”: “Alice”}
{“Name”: “Bob”}
…
INDEXES

Can declare an index on an attribute of a top-most collection

Available:

- **BTREE**: good for equality and range queries
  E.g. name=“Greece”; 20 < age and age < 40
- **RTREE**: good for 2-dimensional range queries
  E.g. 20 < x and x < 40 and 10 < y and y < 50
- **KEYWORD**: good for substring search
SQL++ OVERVIEW

SELECT ... FROM ... WHERE ... [GROUP BY ...]
RETRIEVE EVERYTHING

SELECT x.mondial FROM world x;

Answer

{“mondial”:
   {“country”: [ country1, country2, …],
    “continent”: […],
    “organization”: […],
    ...
    ...
   }
}
Answer

```sql
SELECT x.mondial.country FROM world x;
```

```json
{"country": [ country1, country2, ... ],
}
```
RETRIEVE COUNTRIES, ONE BY ONE

Answer

```
country1
country2
...```
If the value of attribute B is a collection, then we simply iterate over it

SELECT x.A, y.C, y.D
FROM mydata as x, x.B as y;

{x: {a1: {c1: d1, c2: d2}}, a2: {c3: d3}, a3: [{c4: d4}, {c5: d5}]}

x.B is a collection
NESTED COLLECTIONS

If the value of attribute B is a collection, then we simply iterate over it.

```
SELECT x.A, y.C, y.D
FROM mydata as x, x.B as y;
```

```
{"A": "a1", "B": [{"C": "c1", "D": "d1"}, {"C": "c2", "D": "d2"}]}
{"A": "a2", "B": [{"C": "c3", "D": "d3"}]}
{"A": "a3", "B": [{"C": "c4", "D": "d4"}, {"C": "c5", "D": "d5"}]}
```

x.B is a collection

```
{"A": "a1", "C": "c1", "D": "d1"}
{"A": "a1", "C": "c2", "D": "d2"}
{"A": "a2", "C": "c3", "D": "d3"}
{"A": "a3", "C": "c4", "D": "d4"}
{"A": "a3", "C": "c5", "D": "d5"}
```
The problem:

```
{  "mondial":
    {  "country": [ "country1", "country2", ... ],
        "continent": [...],
        "organization": [...],
        ...
    }
}
```

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE y.name='Greece';
```

Runtime error

- city is an array
- city is an object
The problem:

```
{ "mondial":
  { "country": [ country1, country2, ...],
    "continent": [...],
    "organization": [...],
    ...
    ...
  }
}
```

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
     (CASE WHEN z.city is missing THEN []
           WHEN is_array(z.city) THEN z.city
           ELSE [z.city] END) u
WHERE y.name = 'Greece';
```
USEFUL FUNCTIONS

is_array
is_boolean
is_number
is_object
is_string
is_null
is_missing
is_unknown = is_null or is_missing
USEFUL PARADIGMS

Unnesting
Nesting
Group-by / aggregate
Join
Multi-value join
BASIC UNNESTING

An array: [a, b, c]

A nested array: arr = [[a, b], [], [b, c, d]]

Unnest(arr) = [a, b, b, c, d]

SELECT y
FROM arr x, x y
UNNESTING SPECIFIC FIELD

A nested collection

```python
coll = 
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```
UNNESTING SPECIFIC FIELD

A nested collection

coll =

\[
\begin{align*}
{&A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]}, \\
{&A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]}, \\
{&A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]])
\end{align*}
\]

Unnest\textsubscript{\text{F}}(coll) =

\[
\begin{align*}
{&A:a1, B:b1, G:[{C:c1}]}, \\
{&A:a1, B:b2, G:[{C:c1}]}, \\
{&A:a2, B:b3, G[]}, \\
{&A:a2, B:b4, G[]}, \\
{&A:a2, B:b5, G[]}, \\
{&A:a3, B:b6, G:[{C:c2},{C:c3}])}
\end{align*}
\]
UNNESTING SPECIFIC FIELD

A nested collection

coll = 
[A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]], 
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]}, 
{A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Unnest_F(coll) = 
[A:a1, B:b1, G:[{C:c1}]], 
{A:a1, B:b2, G:[{C:c1}]], 
{A:a2, B:b3, G[]}, 
{A:a2, B:b4, G[]}, 
{A:a2, B:b5, G[]}, 
{A:a3, B:b6, G:[{C:c2},{C:c3}]}]

SELECT x.A, y.B, x.G 
FROM coll x, x.F y

Nested Relational Algebra

SQL++

Refers to relations defined on the left
UNNESTING SPECIFIC FIELD

A nested collection

coll =
[A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]],
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ }],
{A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]]]

Unnest_F(coll) =
[A:a1, B:b1, G:[{C:c1}]],
{A:a1, B:b2, G:[{C:c1}]],
{A:a2, B:b3, G[]},
{A:a2, B:b4, G[]},
{A:a2, B:b5, G[]},
{A:a3, B:b6, G:[{C:c2},{C:c3}]]]

SELECT x.A, y.B, x.G
FROM coll x
UNNEST x.F y

Nested Relational Algebra

SQL++
UNNESTING SPECIFIC FIELD

A nested collection

coll =
[A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]],
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
{A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]]]

Unnest_F(coll) =
[A:a1, B:b1, G:[{C:c1}]],
{A:a1, B:b2, G:[{C:c1}]],
{A:a2, B:b3, G[]},
{A:a2, B:b4, G[]},
{A:a2, B:b5, G[]},
{A:a3, B:b6, G:[{C:c2},{C:c3}]]]

Unnest_G(coll) =
[A:a1, F:[{B:b1},{B:b2}], C:c1],
{A:a3, F:[{B:b6}], C:c2},
{A:a3, F:[{B:b6}], C:c3}]

SELECT x.A, y.B, x.G
FROM coll x, x.F y

SELECT x.A, x.F, z.C
FROM coll x, x.G z

Nested Relational Algebra

SQL++
NESTING (LIKE GROUP-BY)

A flat collection

coll = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
NESTING (LIKE GROUP-BY)

A flat collection

\[
\text{coll} = \{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
\]

\[
\text{Nest}_A(\text{coll}) = \{\{A:a1, GRP:\{B:b1, B:b2\}\}, \{A:a2, GRP:\{B:b2\}\}\}
\]

\[
\text{Nest}_B(\text{coll}) = \{\{B:b1, GRP:\{A:a1, A:a2\}\}, \{B:b2, GRP:\{A:a1\}\}\}
\]

Nested Relational Algebra
NESTING (LIKE GROUP-BY)

A flat collection

coll = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

\[
\text{Nest}_A(\text{coll}) = \\
[\{A:a1, \text{GRP:}\{B:b1\},\{B:b2\}\}] \\
[\{A:a2, \text{GRP:}\{B:b2\}\}]
\]

\[
\text{Nest}_B(\text{coll}) = \\
[\{B:b1, \text{GRP:}\{A:a1\},\{A:a2\}\}, \\
\{B:b2, \text{GRP:}\{A:a1\}\}]
\]

SELECT DISTINCT x.A, (SELECT y.B FROM coll y WHERE x.A = y.A) as GRP
FROM coll x
NESTING (LIKE GROUP-BY)

A flat collection

\[
\text{coll} = \{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
\]

Nest\_A(\text{coll}) =
\[
\{\{A:a1, \text{GRP}:[\{B:b1\}, \{B:b2\}]\}\}
\{\{A:a2, \text{GRP}:[\{B:b2\}]\}\}
\]

Nest\_B(\text{coll}) =
\[
\{\{B:b1, \text{GRP}:[\{A:a1\}, \{A:a2\}]\},
\{B:b2, \text{GRP}:[\{A:a1\}]\}\}
\]

SELECT DISTINCT x.A,
(\text{SELECT } y.B \text{ FROM } \text{coll } y \text{ WHERE } x.A = y.A) \text{ as GRP}
FROM \text{coll } x

SELECT DISTINCT x.A, g as GRP
FROM \text{coll } x
\text{LET } g = (\text{SELECT } y.B \text{ FROM } \text{coll } y \text{ WHERE } x.A = y.A)
GROUP-BY / AGGREGATE

A nested collection

coll = 
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
{A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Count the number of elements in the F collection
GROUP-BY / AGGREGATE

A nested collection

coll =

{{A:a1, F:{{B:b1},{B:b2}}, G:{{C:c1}}},
{A:a2, F:{{B:b3},{B:b4},{B:b5}}, G:[]},
{A:a3, F:{{B:b6}}, G:{{C:c2},{C:c3}}}]

Count the number of elements in the F collection

SELECT x.A, COLL_COUNT(x.F) as cnt
FROM coll x
### GROUP-BY / AGGREGATE

<table>
<thead>
<tr>
<th>Function</th>
<th>NULL</th>
<th>MISSING</th>
<th>Empty Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLL_COUNT</td>
<td>counted</td>
<td>counted</td>
<td>0</td>
</tr>
<tr>
<td>COLL_SUM</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_MAX</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_MIN</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_AVG</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_COUNT</td>
<td>not counted</td>
<td>not counted</td>
<td>0</td>
</tr>
<tr>
<td>ARRAY_SUM</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MAX</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MIN</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_AVG</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
</tbody>
</table>
JOIN

Two flat collection

coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
coll2 = [{B:b1,C:c1}, {B:b1,C:c2}, {B:b3,C:c3}]

SELECT x.A, x.B, y.C
FROM coll1 x, coll2 y
WHERE x.B = y.B
MULTI-VALUE JOIN

Recall: a many-to-one relation should have one foreign key, from “many” to “one”

Sometimes people represent it in the opposite direction, from “one” to “many”:

- The reference is a string of keys separated by space
- Need to use `split(string, separator)` to split it into a collection of foreign keys
MULTI-VALUE JOIN

river =
["name": "Donau", "-country": "SRB A D H HR SK BG RO MD UA"],
{"name": "Colorado", "-country": "MEX USA"},
... ]
MULTI-VALUE JOIN

river = 
["name": "Donau", "-country": "SRB A D H HR SK BG RO MD UA"], 
{"name": "Colorado", "-country": "MEX USA"}, 
... ]

split("MEX USA", " ") = 
["MEX", "USA"]
MULTI-VALUE JOIN

river =
["name": "Donau", "-country": "SRB A D H HR SK BG RO MD UA"],
{"name": "Colorado", "-country": "MEX USA"},
...

SELECT ...
FROM country x, river y,
    split(y. `country`, " ") z
WHERE x. `car_code` = z

split("MEX USA", " ") =
["MEX", "USA"]
Query Processing on NFNF data:

Option 1: give up on query plans, use standard java/python-like execution

Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan
FLATTENING SQL++ QUERIES
A nested collection

coll =  
{{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]]}
FLATTENING SQL++
QUERIES

A nested collection

coll =
[A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]],
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
{A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}

coll:
<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
</tr>
</tbody>
</table>

F
<table>
<thead>
<tr>
<th>parent</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

G
<table>
<thead>
<tr>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c1</td>
</tr>
<tr>
<td>3</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>c3</td>
</tr>
</tbody>
</table>
**FLATTENING SQL++ QUERIES**

A nested collection

```
coll = [{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
         {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
         {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

**SQL++**

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

**SQL**

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```
SEMISTRUCTURED DATA MODEL

Several file formats: Json, protobuf, XML

The data model is a tree

They differ in how they handle structure:

- Open or closed
- Ordered or unordered
CONCLUSION

Semistructured data best suited for *data exchange*

For quick, ad-hoc data analysis, use a native query language: SQL++, or AQL, or XQuery

- Modern, advanced query processors like AsterixDB / SQL++ can process semistructured data as efficiently as RDBMS

For long term data analysis: spend the time and effort to normalize it, then store in a RDBMS