Announcements
• Final review session on Wednesday Dec. 11
  • 12:30 - 2:20 pm in SIG 134

Outline
• Unnesting walkthrough
  • Example of exactly what happens

• Artificially nested data
  • .split() operator
  • Joins on .split()

Unnesting General Concept
• Semi-structured data often has nested values
• Bring nested data to the “top level”

Unnesting Play-by-Play
• In SQL:
  1. Cross Products
  2. Selection filters
  3. ...
• SQL++ join proceeds like:
  1. Unnesting
  2. Cross Products
  3. Selection filters
  4. ...
Unnesting Play-by-Play

```
SELECT P.name, O.product
FROM Person AS P, P.orders AS O
UNNEST P.orders AS O
```

Dataset Person (simplified)

```
{{
  {"name": "Dan", "orders": [{"product": "Furby"}]},
  {"name": "Alvin", "orders": [{"product": "Furby"}, {"product": "Magic8"}]},
  {"name": "Magda", "orders": []}
}}
```

For each Person

```
For each

Name  Orders  Product
Dan    Product  Furby
Alvin  Product  Furby  Magic8
Magda  Product  Magic8
```

Name  Orders  Product
Dan    Product  Furby
Alvin  Product  Furby  Magic8
Magda  Product  Magic8
SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

For each Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>Furby</td>
</tr>
<tr>
<td>Magda</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

For each Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>Furby</td>
</tr>
<tr>
<td>Magda</td>
<td>Magic8</td>
</tr>
</tbody>
</table>

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

For each Person

<table>
<thead>
<tr>
<th>Name</th>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan</td>
<td>Furby</td>
</tr>
<tr>
<td>Alvin</td>
<td>Furby</td>
</tr>
<tr>
<td>Magda</td>
<td>Magic8</td>
</tr>
</tbody>
</table>
Unnesting Play-by-Play

For each Person

Name | Orders
--- | ---
Dan | Product: Furby
Alvin | Product: Furby, Magic8
Magda | Product: Furby

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

19

Unnesting Play-by-Play

For each Person

Name | Orders
--- | ---
Dan | Product: Furby
Alvin | Product: Furby, Magic8
Magda | Product: Furby

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

20

Unnesting Play-by-Play

For each Person

Name | Orders
--- | ---
Dan | Product: Furby
Alvin | Product: Furby, Magic8
Magda | Product: Furby

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

21

Unnesting Play-by-Play

For each Person

Name | Orders
--- | ---
Dan | Product: Furby
Alvin | Product: Furby, Magic8
Magda | Product: Furby

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

22

Unnesting Play-by-Play

For each Person

Name | Orders
--- | ---
Dan | Product: Furby
Alvin | Product: Furby, Magic8
Magda | Product: Furby

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

23

Unnesting Play-by-Play

For each Person

Name | Orders
--- | ---
Dan | Product: Furby
Alvin | Product: Furby, Magic8
Magda | Product: Furby

SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O

24
**Unnesting Play-by-Play**

```
SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O
```

For each Person

Name | Orders
--- | ---
Dan | Furby
Alvin | Furby
Magda | Magic8

Magda Product x ∅

**Unnesting Play-by-Play**

```
SELECT P.name, O.product
FROM Person AS P
UNNEST P.orders AS O
```

Name | Orders
--- | ---
Dan | Furby
Alvin | Furby
Magda | Magic8

Dan Furby
Alvin Furby
Magda Product

**Artificial Nesting**

- Collections can be emulated by strings
- Ex: Sentence → Collection of words (tokens)
- Unnesting techniques still apply

```
[ "#NoSQL awesome", "In Codd we trust" ]
```

Nested data:

[ "#NoSQL", "awesome", "In", "Codd", "we", "trust" ]

Unnested:

```
[ "#NoSQL", "awesome", "In", "Codd", "we", "trust" ]
```

**Artificial Nesting**

```
SELECT P.name, field
FROM Person AS P
```

```
SELECT P.name, field
FROM Person AS P, split(P.desc, ")") AS field
```

```
[
  { "name": "Dan", "desc": "DB_Theory" },
  { "name": "Alvin", "desc": "DB_PL" },
  { "name": "Magda", "desc": "DB_Systems" }
]
```

```
[
  { "name": "Dan", "field": "DB_Theory" },
  { "name": "Alvin", "field": "DB_PL" },
  { "name": "Magda", "field": "DB_Systems" }
]
```
Artificial Nesting

SELECT P.name, field
FROM Person AS P, split(P.desc, "_") AS field

1. [Diagram]
   - Don: DB_Theory
   - Alvin: DB_PL
   - Magda: DB_Systems

Artificial Nesting

SELECT P.name, field
FROM Person AS P, split(P.desc, "_") AS field

1. [Diagram]
   - Don: DB_Theory
   - Alvin: DB_PL
   - Magda: DB_Systems

Artificial Nesting

SELECT P.name, field
FROM Person AS P, split(P.desc, "_") AS field

1. [Diagram]
   - Don: DB_Theory
   - Alvin: DB_PL
   - Magda: DB_Systems

Artificial Nesting

SELECT P.name, field
FROM Person AS P, split(P.desc, "_") AS field

1. [Diagram]
   - Don: DB_Theory
   - Alvin: DB_PL
   - Magda: DB_Systems

Artificial Nesting

SELECT P.name, field
FROM Person AS P, split(P.desc, "_") AS field

1. [Diagram]
   - Don: DB_Theory
   - Alvin: DB_PL
   - Magda: DB_Systems

Artificial Nesting

SELECT P.name, field
FROM Person AS P, split(P.desc, "_") AS field

1. [Diagram]
   - Don: DB_Theory
   - Alvin: DB_PL
   - Magda: DB_Systems
Artificial Nesting

```
SELECT P.name, field
FROM Person AS P, split(P.desc, ",") AS field
```

Nesting General Concept

- SQL++ is able to return semi-structured data
- Nesting is similar to the grouping process
- Able to return collections of data for each group

Unnested data:

```
[(x, a), (x, b), (y, a), (y, c), (z, b)]
```

Nested:

```
[(x, [a, b]), (y, [a, c]), (z, [b])] 
```
Nesting Play-by-Play

For each conference, find all people who research the topic

```
SELECT DISTINCT I.cname, people
FROM Interests AS I
LET people = (SELECT VALUE I2.pname
                FROM Interests AS I2
                WHERE I2.topic = I.topic)
```

Final exam topics

- Technically comprehensive, but focus on material after E/R diagrams
- Should still be able to write SQL, RA trees, etc.

Database design

- E/R Diagrams
  - Entity sets, relations, & subclasses
  - Map each to relations
    - multiple ways to do this (many-many, one-many)

Database design

- Constraints
  - key, single-value, referential & other constraints

- Normalization
  - Eliminates anomalies
    - redundancy, update, and deletion anomalies
  - Occur from “bad” functional dependencies (FDs that aren’t superkeys)
  - Apply BCNF decomposition to remove them

3-Tiered Architecture

- B+ tree index is sorted, best for range queries

- clustered vs unclustered
  - clustered always speeds up query
  - but only one index per table can be clustered
  - unclustered only speed up selections if <1% tuples match

Storage & Indexing

For each conference, find all people who research the topic

```
SELECT DISTINCT I.cname, people
FROM Interests AS I
LET people = (SELECT VALUE I2.pname
                FROM Interests AS I2
                WHERE I2.topic = I.topic)
```
Query Optimization

- main cost is disk access I/O
- many logical plans, many physical plans
  - logical plans are RA expressions with desired result
  - physical plans include e.g. choice of join algorithm
  - e.g. block nested loop join and index nest look join
- cost of many operations depends on selectivity
- optimization problem is hard
- realistic goal is to avoid really bad plans

Transactions

- need to provide ACID properties
  - atomic: execute all SQL statements or none
  - consistent: finish with all constraints satisfied
  - isolation: behavior same as if one-at-a-time use
  - durable: committed result are permanent (’til changed)
- consistency maintained by checking constraints
- durability maintained by writing to disk(s)

Transactions II

- isolation achieved through serializable schedules
  - serializable means same behavior as a serial schedule
  - conflict serializable means non-conflicting read/writes can be swapped to make schedule serial
  - stronger than (so implies) serializable
- Strict 2PL
  - 2PL: no unlocks before commit/rollback
  - Strict 2PL: must do all unlocks at commit/rollback
- Need Strict 2PL + predicate locking for full serializability

Semistructured Data Model

- tree structured data: JSON, data is self-describing
  - so schema is not necessary
  - easy to map relation to JSON but not opposite

Relational Parallel Databases

- goal: more nodes => faster or allow more data
  - speed up or scale up
- different architectures
  - shared memory (SQL Server etc.): limited scale
  - shared nothing: really scales (so our focus)
    - basis for parallel processing systems (see previous slides)

Parallel Databases II

- Partition data across nodes (block, hash, range, etc.)
- Query evaluation
  - only one new element: reshuffle
    - move tuples to nodes based on values in certain columns
    - basically same as shuffle sort all MapReduce
    - use to implement all extended RA operations
    - linear speed up or scale up in principle
    - in practice, stragglers are a problem (though see MapReduce)
    - new problem: skewed data
      - may not all fit in memory of one node
Parallel Processing Systems

- MapReduce
  - programming model is one-to-many map function, shuffle sort (grouping), one-to-many reduce function
  - no built-in RA operators
  - but easy to implement since shuffle sort is provided
  - stores intermediate data on disk
  - reasonable if input/output is also to disk (otherwise too slow)

- Spark
  - Built on top of Hadoop with in-memory processing
  - RDD data structure ensures durability

Things NOT on the final exam

- Datalog
- SQL++ queries (but there will be JSON conceptual questions)
- Join algorithms that were not in the slides
  - (no sort-merge joins)
- Hash indexes, you can always assume B+ tree
- Computing query cost in I/O (you will need to know cardinality estimates and computing selectivity)
- Isolation levels (besides serializable)