Welcome

• Welcome to the Special Topics Course: Advanced Topics in Data Management

• Goal: drill deep into modern database engines, new and old techniques, explore extensions.
Staff

• Instructor: Dan Suciu

• TA: Remy Wang
Course Organization

• Lectures: Thursday, 6:30-9:20
  – Guest lecture followed by regular lecture

• Project:
  – Open ended, e.g. extend some query optimizer

• Paper reviews:
  – Short reviews, maybe short programs
  – Please submit *before* the lecture
Evaluation

• Project 50%

• Paper reviews 30%

• Class participation 20%
Communication

• Ed – everybody is subscribed

• Class mailing list – very low traffic

• Website
Lectures

1. Introduction, Review of Query Processing
2. Query Processing (continued)
3. Rebecca Taft (Cockroach Labs)
   Tutorial on Egg (Max Wilsey)
4. Nico Bruno and César A. Galindo-Legaria (Microsoft):
   The Cascades Framework
   Tutorial on optimizing tensor expressions (Remy Wang)
5. Sergey Melnik (Google): Big Query
6. Ippokratis Pandis (Amazon): Redshift
7. Guest lecturer: Doug Brown (Teradata)
8. Guest lecturer: Jiaqi Yan (Snowflake)
9. Guest lecturer: Martin Bravenboer (RelationalAI)
10. Project presentations

Please attend the lectures!
What you (we?) will learn

- Consolidated knowledge of query optimization and execution
- Understand the choices made by various state-of-the-art commercial systems
- Explore possible extensions of optimizers; e.g. to tensor algebra
Tools

• Please have your favorite, state-of-the-art database system on your laptop!
  – Postgres, SQL Server -- yes
  – (access to) Redshift, Snowflake -- yes
  – Sqlite -- no

• The project webpage has links to some open source optimizers
Prerequisites

• If you took csep544, you should be fine in this class

• If you haven’t, then you should think hard if you want to take this class
Today’s Outline

• Overview of SQL processing and optimization
Relational Data Model
Relational Data Model

• A **Database** is a collection of relations

• A **Relation** is a set of tuples
  – Also called **Table**

• A **Tuple** $t$ is an element of $\text{Dom}_1 \times \ldots \times \text{Dom}_n$
Discussion

• Order of records is immaterial

• Sets semantics or bag semantics

• Attribute domains are primitive types: First Normal Form (1NF)
Schema

- **Relation schema**: describes column heads

- **Database schema**: set of all relation schemas
Instance

- **Relation instance**: concrete table content

- **Database instance**: set of relation instances
Relational Query Language

• **Set-at-a-time:**
  – Query inputs and outputs are relations

• **Two variants of the query language:**
  – SQL: declarative
  – Relational algebra: specifies order of operations
Discussion

• **Physical Data Independence:**
  – *No* physical spec of the data

• Declarative query language:
  – Say *what* we want
  – Don’t say *how* to get it

• Query optimization: what → how
SQL
SQL

• Standard query language

• Introduced late 70’s, now it ballooned

• We briefly review “core SQL” (whatever that means)
Structured Query Language: SQL

- **Data definition language: DDL**
  - Statements to create, modify tables and views
  - `CREATE TABLE`, `CREATE VIEW`, `ALTER TABLE`
Structured Query Language: SQL

• **Data definition language: DDL**
  – Statements to create, modify tables and views
  – CREATE TABLE …,
    CREATE VIEW …,
    ALTER TABLE…

• **Data manipulation language: DML**
  – Statements to issue queries, insert, delete data
  – SELECT-FROM-WHERE…,
    INSERT…,
    UPDATE…,
    DELETE…

  Our focus
SQL Query

Basic form: (plus many many more bells and whistles)

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```
Quick Review of SQL

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, qty, price)
Part(pno, pname, psize, pcolor)
Quick Review of SQL

Retrieve all parts under $100, and the cities in Washington that supply them:
Quick Review of SQL

Retrieve all parts under $100, and the cities in Washington that supply them:

```
SELECT DISTINCT z.pno, z.pname, x.scity
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno
    and y.pno = z.pno
    and x.sstate = 'WA'
    and y.price < 100
```
Terminology

• **Selection/filter**: return a subset of the rows:
  – SELECT * FROM Supplier
    WHERE scity = 'Seattle'

• **Projection**: return subset of the columns:
  – SELECT DISTINCT scity FROM Supplier;

• **Join**: refers to combining two or more tables
  – SELECT * FROM Supplier, Supply, Part …
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```sql
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno
```
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno

Need TWO Suppliers and TWO Supplies
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno

Need TWO Suppliers and TWO Supplies
one in Seattle
the other in Portland

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland
Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
  and x1.sno = y1.sno
  and x2.scity = 'Portland'
  and x2.sno = y2.sno
  and y1.pno = y2.pno
```

Need TWO Suppliers and TWO Supplies

one in Seattle
the other in Portland

the SAME part
Nested-Loop Semantics of SQL

```
SELECT [DISTINCT] a_1, a_2, ..., a_k
FROM   R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE  Conditions
```
Nested-Loop Semantics of SQL

```
SELECT [DISTINCT] a_1, a_2, ..., a_k
FROM   R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE  Conditions

Answer = {} 
for x_1 in R_1 do 
  for x_2 in R_2 do 
    ..... 
    for x_n in R_n do 
      if Conditions 
      then Answer = Answer ∪ {(a_1, ..., a_k)} 
return Answer
```
Nested-Loop Semantics of SQL

```
SELECT [DISTINCT] a_1, a_2, ..., a_k
FROM   R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE  Conditions

Answer = {}
for x_1 in R_1 do
    for x_2 in R_2 do
        ..... 
        for x_n in R_n do
            if Conditions
                then Answer = Answer \cup \{ (a_1, ..., a_k) \}

return Answer
```

This SEMANTICS!
It is NOT how the engine computes the query!
Query Evaluation
Query Evaluation

• Convert SQL into a query plan

• Optimize the query plan

• Execute each operator of the query plan
Relational algebra (subset)

• Selection $\sigma_{\text{condition}}$

• Projection $\Pi_{\text{attributes}}$

• Join $\bowtie_{\text{condition}}$

• Duplicate elimination $\delta$
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno
Convert SQL to Query Plan

```
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno
```
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
and x1.sno = y1.sno
and x2.scity = 'Portland'
and x2.sno = y2.sno
and y1.pno = y2.pno

\sigma_{x1.city='Seattle'} \bowtie x1.sno=y1.sno
\sigma_{x1.city='Portland'} \bowtie x2.sno=y2.sno

Supplier x1  Supply y1  Supplier x2  Supply y2
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
  and x1.sno = y1.sno
  and x2.scity = 'Portland'
  and x2.sno = y2.sno
  and y1.pno = y2.pno

Supplier x1
⋈
Supply y1
⋈
Supplier x2
⋈
Supply y2

\( \delta \)

\( \Pi_{y1.pno} \)

\( \Join_{y1.pno=y2.pno} \)

\( \sigma_{x1.city='Seattle'} \)

\( \sigma_{x1.city='Portland'} \)

\( \Join_{x1.sno=y1.sno} \)

\( \Join_{x2.sno=y2.sno} \)
Convert SQL to Query Plan

```
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno
```

Sometimes we assume that \( \Pi \) already eliminates duplicates; no need for \( \delta \)
Optimize the Query Plan

• Heuristics:
  – Push selections down
  – Pull projections up

• Cost based:
  – Join reordering: dynamic programming
  – Rule based
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno
SELECT DISTINCT  y1.pno
FROM  Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE  x1.scity = 'Seattle'
        and x1.sno = y1.sno
        and x2.scity = 'Portland'
        and x2.sno = y2.sno
        and y1.pno = y2.pno
SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno

...and Pull Projections Up

\[ \delta \]

\[ \Pi_{y1.pno} \]

\[ \Join_{y1.pno=y2.pno} \]

\[ \Join_{x1.sno=y1.sno} \]

\[ \sigma_{x1.city='Seattle'} \]

Supplier x1

\[ \Join_{x2.sno=y2.sno} \]

\[ \sigma_{x2.city='Portland'} \]

Supplier x1

Supply y1

Supplier x1

Supply y1
Optimize the Query Plan

• Heuristics:
  – Push selections down
  – Pull projections up

• Cost based:
  – Join reordering: dynamic programming
  – Rule based
Join Reordering

SELECT DISTINCT y1.pno
FROM Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE x1.scity = 'Seattle'
    and x1.sno = y1.sno
    and x2.scity = 'Portland'
    and x2.sno = y2.sno
    and y1.pno = y2.pno

Optimize this part
Joins Reordering

• It’s the bread and butter of query optimizers

• Performed using dynamic programming, a.k.a. Selinger’s algorithm

• Before we see this, let’s examine how joins are evaluated
Join Evaluation Algorithms

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

Three algorithms:
1. Nested Loops
2. Hash-join
3. Merge-join
1. Nested Loop Join

Logical operator:
\[ \text{Supplier} \bowtie_{\text{sid}=\text{sid}} \text{Supply} \]

for \( x \) in Supplier do
    for \( y \) in Supply do
        if \( x.\text{sid} = y.\text{sid} \) then
            output\((x,y)\)

Runtime \( O(n^2) \)
2. Hash Join

Logical operator:

Supply $\bowtie_{\text{sid} = \text{sid}}$ Supplier

for x in Supplier do
  insert(x.sid, x)
2. Hash Join

Logical operator:

\[ \text{Supply} \bowtie_{\text{sid}=\text{sid}} \text{Supplier} \]

for \( x \) in \text{Supplier} do
\[
\text{insert}(x.\text{sid}, x)
\]

for \( y \) in \text{Supply} do
\[
x = \text{find}(y.\text{sid});
\]
\[
\text{output}(x, y);
\]
2. Hash Join

Logical operator:
Supply $\bowtie_{sid=sid}$ Supplier

for $x$ in Supplier do
   insert($x$.sid, $x$)
for $y$ in Supply do
   $x = \text{find}(y$.sid$)$;
   output($x,y$);

Runtime $O(n)$
2. Hash Join

Logical operator:

\[ \text{Supplier} \Join_{\text{sid} = \text{sid}} \text{Supply} \]

for y in Supply do
insert(y.sid, y)

for x in Supplier do ??

Supplier\((\text{sid}, \text{sname}, \text{scity}, \text{sstate})\)
Supply\((\text{sid}, \text{pno}, \text{quantity})\)
2. Hash Join

Logical operator:
Supplier \( \bowtie_{sid=sid} \) Supply

for y in Supply do
   insert(y.sid, y)

for x in Supplier do
   for y in find(x.sid) do
      output(x,y);
2. Hash Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid}=\text{sid}} \text{Supply} \]

for \( y \) in \text{Supply} do
    insert(\( y.\text{sid}, y \))

for \( x \) in \text{Supplier} do
    for \( y \) in find(\( x.\text{sid} \)) do
        output(\( x,y \));

Runtime can be \( O(n^2) \) because \text{Supply.sid} is not a key and there may be many duplicates.
3. Merge Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
3. Merge Join

Logical operator:

\[
\text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply}
\]

Sort(Supplier); Sort(Supply);
\[x = \text{Supplier}.\text{first}();\]
\[y = \text{Supply}.\text{first}();\]
while \(y \neq \text{NULL} \) do

case:
- \(x.\text{sid} < y.\text{sid}: \text{???}\)
- \(x.\text{sid} = y.\text{sid}: \text{???}\)
- \(x.\text{sid} > y.\text{sid}: \text{???}\)
3. Merge Join

Logical operator:

\[
\text{Supplier} \Join_{\text{sid}=\text{sid}} \text{Supply}
\]

Sort(Supplier); Sort(Supply);
\[
x = \text{Supplier}.\text{first}();
\]
\[
y = \text{Supply}.\text{first}();
\]
while \(y \neq \text{NULL}\) do
  case:
    \(x.\text{sid} < y.\text{sid}\): \(x = x.\text{next}()\)
    \(x.\text{sid} = y.\text{sid}\): ???
    \(x.\text{sid} > y.\text{sid}\): ???
3. Merge Join

Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

Sort(Supplier); Sort(Supply);

\[
\begin{align*}
x &= \text{Supplier}.\text{first}(); \\
y &= \text{Supply}.\text{first}(); \\
\text{while } y \neq \text{NULL do} & \\
\quad \text{case:} & \\
\quad \quad x.\text{sid} < y.\text{sid}: & x = x.\text{next}() \\
\quad \quad x.\text{sid} = y.\text{sid}: & \text{output}(x,y); y = y.\text{next}(); \\
\quad \quad x.\text{sid} > y.\text{sid}: & \text{???}
\end{align*}
\]
3. Merge Join

Logical operator:

\[ \text{Supplier} \Join_{\text{sid} = \text{sid}} \text{Supply} \]

Sort(Supplier); Sort(Supply);
\( x = \text{Supplier}.\text{first}(); \)
\( y = \text{Supply}.\text{first}(); \)
while \( y \neq \text{NULL} \) do
  case:
    \( x.\text{sid} < y.\text{sid} \): \( x = x.\text{next}() \)
    \( x.\text{sid} = y.\text{sid} \): output(\( x, y \)); \( y = y.\text{next}() \)
    \( x.\text{sid} > y.\text{sid} \): \( y = y.\text{next}() \)
3. Merge Join

Logical operator:

Supplier \( \bowtie_{\text{sid} = \text{sid}} \) Supply

Sort(Supplier); Sort(Supply);
\( x = \text{Supplier} . \text{first}(); \)
\( y = \text{Supply} . \text{first}(); \)
while \( y \neq \text{NULL} \) do
  case:
  \( x . \text{sid} < y . \text{sid} \): \( x = x . \text{next}() \)
  \( x . \text{sid} = y . \text{sid} \): output(\(x,y\)); \( y = y . \text{next}() \)
  \( x . \text{sid} > y . \text{sid} \): \( y = y . \text{next}() \)

Runtime \( O(n \log(n)) \) (because sorting…)

Supplier(\text{sid}, \text{sname}, \text{scity}, \text{sstate})
Supply(\text{sid}, \text{pno}, \text{quantity})
Discussion

• Joins = most studied relational operator

• Variations:
  – Blocking (materialize) v.s. pipelining
  – Main memory join v.s. external memory
  – Single server v.s. distributed
Join Ordering
Optimize the Query Plan

- **Heuristics:**
  - Push selections down
  - Pull projections up

- **Cost based:**
  - Join reordering: dynamic programming
  - Rule based
Join Reordering

- Dynamic programming

- Introduced by Selinger, “System R”, 79

- Also called Selinger’s algorithm

- Originally restricted to:
  - Left-deep plans
  - No cartesian products
Cartesian Products

\( R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D) \)
Cartesian Products

\[ R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D) \]
Cartesian Products

\[ R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D) \]
Cartesian Products

\[ R(A, B) \bowtie_{R.B = S.B} S(B, C) \bowtie_{S.C = T.C} T(C, D) \]

Without cartesian product

\[ R(A, B) \bowtie_{R.B = S.B} S(B, C) \bowtie_{S.C = T.C} T(C, D) \]
Cartesian Products

\[ R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D) \]

Without cartesian product

\[ R(A,B) \bowtie S(B,C) \bowtie T(C,D) \]

With cartesian product

\[ R(A,B) \bowtie S(B,C) \bowtie T(C,D) \]
Cartesian Products

\[ R(A,B) \bowtie_{R.B=S.B} S(B,C) \bowtie_{S.C=T.C} T(C,D) \]

Without cartesian product

With cartesian product

When could this plan be better?
Shapes of Join Trees

Left deep

\[ R_1 \bowtie R_2 \bowtie R_{n-1} \bowtie R_n \]
Shapes of Join Trees

Left deep

\[ \ldots \]

Hash-tables built on right relations
Shapes of Join Trees

Left deep

Right deep

Hash-tables built on right relations
Shapes of Join Tree Shapes

Left deep

Right deep

Bushy

Hash-tables built on right relations
Shapes of Join Trees

Left deep

... R_{n-1} R_n

R_1 R_2

Bushy

Zig-zag

Hash-tables built on right relations

Right deep

R_1 R_2 R_3 ...

R_1 R_2 R_3 ...

R_4 R_5 ...

Dynamic Programming

• Join order: a misnomer, since we are not just ordering, but we compute a tree

• Main idea: compute optimal join order for every subset of relations

• With or without cartesian products
  With or without restricting tree shapes
Dynamic Programming

• Let m = number of relations to join
• For s = 1, m do:
  – For each subset S of of size s do:
    • Split S into [relation R] + [set of s-1 relations S’]
    • Lookup Cost(S’)
    • Cost(S) := \( \min_{splits} (Cost(S’) + \text{cost-of}(R \bowtie S’)) \)
    • Memorize (S, Cost(S))
• Return Cost(All-relations)
Discussion

• Dynamic programming: exponential in # of relations; works for up to 10-20 rels

• Variations:
  – “Interesting orders” for merge-join
  – With or without cartesian product
  – Left-, right-, bushy-, zig-zag plans
  – Outerjoins? Anti-semijoins?
NULLs in SQL
NULLs in SQL

- A NULL value means missing, or unknown, or undefined, or inapplicable
NULLs in WHERE Clause

A **WHERE** clause contains a predicate:
- • Expr1 op Expr2

How do we compute the predicate when values are NULL?

Example

```sql
where price < 100 and (pcolor='red' or psize=2)
```
SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1
SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- $A = B$, $A < B$, …: Unknown, if either $A$ or $B$ is NULL
  AND, OR, NOT: min, max, and 1- …
SQL Has Three-Valued Logic

• False=0, Unknown=0.5, True=1

• A = B, A < B, …: **Unknown**, if either A or B is NULL
  AND, OR, NOT: **min**, **max**, and 1- …

• Return only tuples whose condition is True
SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- A = B, A < B, …: *Unknown*, if either A or B is NULL
  AND, OR, NOT: *min*, *max*, and 1- …

- Return only tuples whose condition is *True*

- E.g. price < 100: can be False, Unknown, or True
SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- A = B, A < B, …: Unknown, if either A or B is NULL
  AND, OR, NOT: min, max, and 1- …

- Return only tuples whose condition is True

- E.g. price < 100: can be False, Unknown, or True

- What about (price < 100) and (pcolor = ‘red’)?
SQL Has Three-Valued Logic

```sql
select * 
from Part
where price < 100 
and (psize=2 or pcolor='red')
```

<table>
<thead>
<tr>
<th>pno</th>
<th>pname</th>
<th>price</th>
<th>psize</th>
<th>pcolor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iPad</td>
<td>500</td>
<td>13</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>Scooter</td>
<td>99</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>Charger</td>
<td>NULL</td>
<td>NULL</td>
<td>red</td>
</tr>
<tr>
<td>4</td>
<td>iPad</td>
<td>50</td>
<td>2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

(in class: discuss which tuples are returned)
SQL Has Three-Valued Logic

Problem: does *not* return all records!

```
select *
from Part
where (price <= 100) or (price > 100)
```
SQL Has Three-Valued Logic

Problem: does \textit{not} return all records!

\begin{verbatim}
select *
from Part
where (price \leq 100) or (price > 100)
\end{verbatim}

Now it does

\begin{verbatim}
select *
from Part
where (price \leq 100) or (price > 100) or isNull(price)
\end{verbatim}
NULLs and their 3-valued logic are a major headache for query optimizers:

- \((A \text{ and not}(A)) \neq \text{True}\)
- Aggregates need special cases
- Outerjoins are not commutative, etc
Aggregates in SQL
Aggregates

```
SELECT count(*)
FROM Part
```
For each city, compute the average size of parts supplied from that city.

```
SELECT count(*)
FROM Part
```
Aggregates

For each city, compute the average size of parts supplied from that city.

```
SELECT count(*)
FROM Part
```

```
SELECT x.scity, avg(psize)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY x.scity
```
Aggregates

For each city, compute the average size of parts supplied from that city.

...but only for cities that supply > 200 parts
SELECT count(*)
FROM Part

SELECT x.scity, avg(psize)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY x.scity

SELECT x.scity, avg(psize)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY x.scity
HAVING count(*) > 200

For each city, compute the average size of parts supplied from that city.

...but only for cities that supply > 200 parts
Aggregates

• Semantics:
  – FROM-WHERE (nested-loop semantics)
  – Group answers by GROUP BY attrs
  – Apply HAVING predicates on groups
  – Apply SELECT aggregates on groups

• Aggregate functions:
  – count, sum, min, max, avg
Relational Algebra

- Group-by:

\[ \gamma \text{attributes}, \text{agg}(A_1) \rightarrow B_1, \text{agg}(A_2) \rightarrow B_2, \ldots \]
Rule-based Optimization

• Collection of rewrite rules:
  \[ E_1 = E_1' \]
  \[ E_2 = E_2' \]
  ...

• Given a query plan \( P \), apply rules repeatedly, to generate equivalent plans:
  \[ P = P_1 = P_2 = P_3 = \ldots \]

• Return the plan with lowest cost
Examples of rules

\[(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)\]

\[\gamma(R \bowtie S) = \gamma(R \bowtie \gamma(S))\]

\[\gamma(R \cup S) = \gamma(\gamma(R) \cup \gamma(S))\]
Aggregate
Push-down

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

SELECT x.sstate, sum(y.quantity * z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
SELECT x.sstate, sum(y.quantity*z.price) 
FROM Supplier x, Supply y, Part z 
WHERE x.sid = y.sid and y.pno = z.pno 
GROUP BY x.sstate
Aggregate Push-down

\[ \gamma_{x.\text{sstate}}, \sum(y.\text{quantity} \cdot z.\text{price}) \]

\[ \bowtie_{x.\text{sid} = y.\text{sid}} \]

\[ \bowtie_{y.\text{pno} = z.\text{pno}} \]

\[ \text{Supplier } x \]
\[ \text{Supply } y \]
\[ \text{Part } z \]

\[ \text{SELECT } x.\text{sstate}, \sum(y.\text{quantity} \cdot z.\text{price}) \]
\[ \text{FROM Supplier } x, \text{ Supply } y, \text{ Part } z \]
\[ \text{WHERE } x.\text{sid} = y.\text{sid} \text{ and } y.\text{pno} = z.\text{pno} \]
\[ \text{GROUP BY } x.\text{sstate} \]
Discussion

• Rule-based optimizer introduced by Graefe in the Volcano system, at Wisconsin

• Later refined by Graefe into the CASCADES framework → SQL Server

• Most modern systems use rule-based optimizers

• EGG = open-source equality saturation system
Outer Joins
Outer joins

Compute the number of products sold by each supplier

```
SELECT x.sno, x.sname, count(*)
FROM Supplier x, Supply y
WHERE x.sno = y.sno
GROUP BY x.sno, x.sname
```

Problem: suppliers with 0 products are not included.
Compute the number of products sold by each supplier

```
SELECT x.sno, x.sname, count(y.sno)
FROM Supplier x LEFT OUTER JOIN Supply y
    ON x.sno = y.sno
GROUP BY x.sno, x.sname
```

Now they are included
Left Outer Join (Details)

from R left outer join S on C1 where C2

1. Compute cross product $R \times S$

2. Filter on C1

3. Add all R records without a match

4. Filter on C2
Joins

- **Inner join** = includes only matching tuples (i.e. regular join)
- **Left outer join** = includes everything from the left
- **Right outer join** = includes everything from the right
- **Full outer join** = includes everything
Relational Algebra

• Left outer join: $\bowtie$

• Right outer join: $\bowtie$

• Full outer join: $\bowtie$
Hash-based Left Outer Join

**Supplier** \(\bowtie_{\text{sid} = \text{sid}}\) **Supply**

```plaintext
for x in Supplier do
    insert(x.sid, x)

for y in Supply do
    x = find(y.sid);
    y.found = true
    output(x,y);

for x in Supplier do
    if not x.found
        then output(x,NULL)
```

**Supplier** \(\text{sid}, \text{sname}, \text{scity}, \text{sstate})
**Supply** \(\text{sid}, \text{pno}, \text{quantity})

Discussion

• Left outer join:
  – Very useful for one-to-many relationships
  – Eg each Supplier has 0 or more Supply
  – Eg each Student takes 0 or more Courses

• Right outer join, full outer join: rarely used

• Major pain for optimization
Subqueries in SQL
Subqueries

- Subquery in SELECT:
  - Must return single value
- Subquery in FROM
  - Like a temporary relation
  - Alternative: use the WITH clause
- Subquery in WHERE or in HAVING
  - Can express sophisticated queries
Subquery in SELECT

Compute the number of products sold by each supplier

```
SELECT x.sno, x.sname, 
   (SELECT count(*)
        FROM Supply y
       WHERE x.sno = y.sno)
FROM Supplier x
```
Subquery in FROM

Better: use the WITH statement!
Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts
Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
FROM Supplier x, Supply y
WHERE x.sno = y.sno
GROUP BY x.sno)

```
WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
FROM Supplier x, Supply y
WHERE x.sno = y.sno
GROUP BY x.sno)
```

For each supplier, compute how many parts they supply
Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
FROM Supplier x, Supply y
WHERE x.sno = y.sno
GROUP BY x.sno),
Mx AS (SELECT max(c) as m
FROM Cnt)
Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
  FROM Supplier x, Supply y
  WHERE x.sno = y.sno
  GROUP BY x.sno),
  Mx AS (SELECT max(c) as m
  FROM Cnt)
SELECT z.sno, z.sname, m.m
FROM Cnt z, Mx m
WHERE z.c = m.m;

Find the "witness", i.e. the supplier that supplies the maximum number of parts; argmax
Subquery in WHERE

Find suppliers that supply some ‘blue’ parts
Find suppliers that supply some ‘blue’ parts

```
SELECT x.sno 
FROM Supplier x 
WHERE exists (SELECT * FROM Supply y, Part z 
WHERE x.sno=y.sno 
    and y.pno=z.pno 
    and z.pcolor = 'blue');
```
Subquery in WHERE

Find suppliers that supply *only* ‘red’ parts
Subquery in WHERE

Find suppliers that supply *only* ‘red’ parts

Find the *other* suppliers
Find suppliers that supply *only* ‘red’ parts

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
  WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor != 'red');
```

Find the *other* suppliers
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

**Subquery in WHERE**

Find suppliers that supply *only* ‘red’ parts

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
              and y.pno=z.pno
              and z.pcolor != 'red');
```

Find the *other* suppliers

```
SELECT x.sno
FROM Supplier x
WHERE not exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
              and y.pno=z.pno
              and z.pcolor != 'red');
```

Negate to get the right ones
Relational Algebra

• Semijoin: $R \bowtie S$
  – Subset of $R$ that joins with $S$
  – $R \bowtie S = \Pi_{\text{Attrs}(R)}(R \bowtie S)$

• Anti-semijoin: $R \triangleright S$
  – Subset of $R$ that does not join with $S$
  – $R \triangleright S = R - (R \bowtie S)$
Semi-Join

Find suppliers that supply some ‘blue’ parts

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
  WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor = 'blue');
```
Semi-Join

Find suppliers that supply some ‘blue’ parts

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
WHERE x.sno=y.sno
and y.pno=z.pno
and z.pcolor = 'blue');
```
Anti-semi-Join

Find suppliers that supply *only* ‘red’ parts

```sql
SELECT x.sno
FROM Supplier x
WHERE not exists (SELECT * FROM Supply y, Part z
    WHERE x.sno=y.sno
    and y.pno=z.pno
    and z.pcolor != 'red');
```
Anti-semi-Join

Find suppliers that supply *only* ‘red’ parts

\[
\text{SELECT } x\text{.sno} \\
\text{FROM Supplier } x \\
\text{WHERE not exists (SELECT * FROM Supply y, Part z} \\
\text{WHERE x\text{.sno}=y\text{.sno} } \\
\text{and y\text{.pno}=z\text{.pno} } \\
\text{and z\text{.pcolor} \\! = ‘red’);}
\]
Discussion

• RA does not have variables
  – Exception: “dependent” join allows variables, but needs to be removed

• Query unnesting: rewriting a query with subqueries into a query without subqueries

• Some systems fail to unnest complicated queries: nested loop join
Operator Interface
How Do We Combine Them?
How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops
How Do We Combine Them?

Option 1: materialize intermediate results

Option 2: Pipeline tuples btw. ops

Implementation: Iterator Interface
Operator Interface

Volcano model:
• `open()`, `next()`, `close()`
• Pull model
• Volcano optimizer: G. Graefe’s (Wisconsin) → SQL Server
• Supported by most DBMS today
• Will discuss next
Operator Interface

Volcano model:
• `open()`, `next()`, `close()`
• Pull model
• Volcano optimizer: G. Graefe’s (Wisconsin) → SQL Server
• Supported by most DBMS today

Data-driven model:
• `open()`, `produce()`, `consume()`, `close()`
• Push model
• Introduced by Thomas Neumann in Hyper (at TU Munich), later acquired by Tableau
Discussion

• Most systems adopt the Volcano-model, a.k.a. the iterator interface
• Vectorized processing = iterator interface that processes a block of tuples (vector?) instead of one tuple
• Compiled model = compile to machine code and use the push model