Chapter 16: Query Compiler
Query Compiler

Parsing
Logical Query Plan
SQL query

parse

parse tree

convert

logical query plan

apply laws

“improved” l.q.p

estimate result sizes

l.q.p. +sizes

consider physical plans

\{P_1, P_2, \ldots \}

pick best

\{(P_1, C_1), (P_2, C_2), \ldots \}

estimate costs

execute

answer

statistics

Pi
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Example: SQL query

```
SELECT title
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);

(Find the movies with stars born in 1960)
```

Assume we have a simplified grammar for SQL.
Example: Parse Tree

```
<Query>
  <SFW>
    SELECT <SelList> FROM <FromList> WHERE <Condition>
    <Attribute> <RelName> <Tuple> IN <Query>
    title StarsIn starName
  <SFW>
    SELECT <SelList> FROM <FromList> WHERE <Condition>
    <Attribute> <RelName> <Attribute> LIKE <Pattern>
    name MovieStar birthDate '1960'
```
The Preprocessor

- replaces each reference to a view with a parse (sub)-tree that describes the view (i.e., a query)
- does semantic checking:
  - are relations and views mentioned in the schema?
  - are attributes mentioned in the current scope?
  - are attribute types correct?
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Convert Parse Tree to Relational Algebra

- Complete algorithm depends on specific grammar, which determines forms of the parse trees
- Here give a flavor of the approach
Conversion

- Suppose there are no subqueries.

- SELECT \(att\text{-}list\) FROM \(rel\text{-}list\) WHERE \(cond\)

  is converted into

  \[
  \text{PROJ}_{att\text{-}list}(\text{SELECT}_{cond}(\text{PRODUCT}(rel\text{-}list))), \text{ or }
  \]

  \[
  \pi_{att\text{-}list}(\sigma_{cond}(X(rel\text{-}list)))
  \]
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND birthdate LIKE '%1960';
Equivalent Algebraic Expression Tree

\[ \pi_{\text{movieTitle}} \]

\[ \sigma \text{ starname} = \text{name AND birthdate LIKE '1960'} \]

\[ X \]

\[ \text{StarsIn} \quad \text{MovieStar} \]
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- **Improve the logical query plan**
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Improving the Logical Query Plan

- There are numerous algebraic laws concerning relational algebra operations.
- By applying them to a logical query plan judiciously, we can get an equivalent query plan that can be executed more efficiently.
- Next we'll survey some of these laws.
Selections usually reduce the size of the relation
Usually good to do selections early, i.e., "push them down the tree"
Also can be helpful to break up a complex selection into parts
Selection Splitting

\[ \sigma_{C_1 \text{ AND } C_2} (R) = \sigma_{C_1} (\sigma_{C_2} (R)) \]

\[ \sigma_{C_1 \text{ OR } C_2} (R) = (\sigma_{C_1} (R)) \cup_{\text{set}} (\sigma_{C_2} (R)) \]

if \( R \) is a set

\[ \sigma_{C_1} (\sigma_{C_2} (R)) = \sigma_{C_2} (\sigma_{C_1} (R)) \]
Selection and Binary Operators

- Must push selection to both arguments:
  - $\sigma_C (R \cup S) = \sigma_C (R) \cup \sigma_C (S)$

- Must push to first arg, optional for 2nd:
  - $\sigma_C (R - S) = \sigma_C (R) - S$
  - $\sigma_C (R - S) = \sigma_C (R) - \sigma_C (S)$

- Push to at least one arg with all attributes mentioned in C:
  - product, natural join, theta join, intersection
  - e.g., $\sigma_C (R \times S) = \sigma_C (R) \times S$, if R has all the atts in C
Pushing Selection *Up* the Tree

- Suppose we have relations
  - StarsIn(title, year, starName)
  - Movie(title, year, len, inColor, studioName)
- and a view
  - CREATE VIEW MoviesOf1996 AS
    - SELECT *
    - FROM Movie
    - WHERE year = 1996;
- and the query
  - SELECT starName, studioName
    FROM MoviesOf1996 NATURAL JOIN StarsIn;
The Straightforward Tree

\[ \pi_{\text{starName}, \text{studioName}} \]

\[ \sigma_{\text{year}=1996} \]

StarsIn

Movie

Remember the rule
\[ \sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S \]?
The Improved Logical Query Plan

\[ \pi_{\text{starName, studioName}} \sigma_{\text{year}=1996} \text{StarsIn} \]

\[ \pi_{\text{starName, studioName}} \sigma_{\text{year}=1996} \]
Laws Involving Projections

- Consider adding in additional projections
- Adding a projection lower in the tree can improve performance, since often tuple size is reduced
  - Usually not as helpful as pushing selections down
- If a projection is inserted in the tree, then none of the eliminated attributes can appear above this point in the tree
  - Ex: \( \pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S)) \), where \( M \) (resp. \( N \)) is all attributes of \( R \) (resp. \( S \)) that are used in \( L \)
- Another example:
  - \( \pi_L(R \cup_{\text{bag}} S) = \pi_L(R) \cup_{\text{bag}} \pi_L(S) \)
  - But watch out for set union!
Push Projection Below Selection?

- Rule: $\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$
  where $M$ is all attributes used by $L$ or $C$
- But is it a good idea?

SELECT starName FROM StarsIn WHERE movieYear = 1996;

\[
\begin{array}{c}
\pi \\
\sigma \\
\end{array}
\begin{array}{c}
\pi \\
\sigma \\
\pi \\
\end{array}
\begin{array}{c}
\text{starName} \\
\text{movieYear=1996} \\
\text{starName, movieYear} \\
\end{array}
\begin{array}{c}
\text{starName} \\
\text{movieYear=1996} \\
\text{StarsIn} \\
\end{array}
\begin{array}{c}
\text{starName} \\
\text{movieYear=1996} \\
\text{StarsIn} \\
\end{array}
\]
Joins and Products

- Recall from the definitions of relational algebra:
  - \( R \bowtie_C S = \sigma_C(R \times S) \) (theta join)
  - \( R \bowtie S = \pi_L(\sigma_C(R \times S)) \) (natural join)
    where \( C \) equates same-name attributes in \( R \) and \( S \), and \( L \) includes all attributes of \( R \) and \( S \) dropping duplicates

- To improve a logical query plan, replace a product followed by a selection with a join
  - Join algorithms are usually faster than doing product followed by selection
Duplicate Elimination

- Moving δ down the tree is potentially beneficial as it can reduce the size of intermediate relations
- Can be eliminated if argument has no duplicates
  - a relation with a primary key
  - a relation resulting from a grouping operator
- Legal to push δ through product, join, selection, and bag intersection
  - Ex: δ(R X S) = δ(R) X δ(S)
- Cannot push δ through bag union, bag difference or projection
Grouping and Aggregation

Since $\gamma$ produces no duplicates:

$\delta(\gamma_L(R)) = \gamma_L(R)$

Get rid of useless attributes:

$\gamma_L(R) = \gamma_L(\pi_M(R))$

where $M$ contains all attributes in $L$

If $L$ contains only MIN and MAX:

$\gamma_L(R) = \gamma_L(\delta(R))$
Example

- Suppose we have the relations
  - MovieStar(name, addr, gender, birthdate)
  - StarsIn(title, year, starName)

- and we want to find the youngest star to appear in a movie for each year:
  \[
  \text{SELECT year, MAX(birthdate)} \\
  \text{FROM MovieStar, StarsIn} \\
  \text{WHERE name = starName} \\
  \text{GROUP BY year;}
  \]
Example cont'd

\[ \gamma_{\text{year,MAX(birthdate)}} \]

\[ \sigma_{\text{name=starName}} \]

\[ \chi \]

MovieStar  \rightarrow  StarsIn

\[ \delta \]

MovieStar  \rightarrow  StarsIn

\[ \pi_{\text{year,birthdate}} \]

\[ \pi_{\text{year,birthdate}} \]

\[ \pi_{\text{year,birthdate}} \]

\[ \pi_{\text{year,starName}} \]
Summary of LQP Improvements

- Selections:
  - push down tree as far as possible
  - if condition is an AND, split and push separately
  - sometimes need to push up before pushing down

- Projections:
  - can be pushed down
  - new ones can be added (but be careful)

- Duplicate elimination:
  - sometimes can be removed

- Selection/product combinations:
  - can sometimes be replaced with join
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Grouping Assoc/Comm Operators

- Group together adjacent joins, adjacent unions, and adjacent intersections as siblings in the tree.
- Sets up the logical QP for future optimization when physical QP is constructed: determine best order for doing a sequence of joins (or unions or intersections).
Evaluating Logical Query Plans

- The transformations discussed so far intuitively seem like good ideas.
- But how can we evaluate them more scientifically?
- Estimate size of relations, also helpful in evaluating physical query plans.
- Coming up next…
Query Compilation

Evaluating Logical Query Plan
Physical Query Plan
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Estimating Sizes of Relations

- Used in two places:
  - to help decide between competing logical query plans
  - to help decide between competing physical query plans

- Notation review:
  - $T(R)$: number of tuples in relation $R$
  - $B(R)$: minimum number of block needed to store $R$
  - $V(R,a)$: number of distinct values in $R$ of attribute $a$
Desiderata for Estimation Rules

1. Give accurate estimates
2. Are easy (fast) to compute
3. Are logically consistent: estimated size should not depend on *how* the relation is computed

Here describe some simple heuristics.

All we really need is a scheme that properly *ranks* competing plans.
Estimating Size of Projection

- This can be exactly computed
- Every tuple changes size by a known amount.
Estimating Size of Selection

Suppose selection condition is \( A = c \), where \( A \) is an attribute and \( c \) is a constant.

A reasonable estimate of the number of tuples in the result is:

\[ \frac{T(R)}{V(R,A)} \]

i.e., original number of tuples divided by number of different values of \( A \)

Good approximation if values of \( A \) are evenly distributed

Also good approximation in some other, common, situations (see textbook)
Estimating Size of Selection (cont'd)

- If condition is $A < c$:
  - a good estimate is $T(R)/3$; intuition is that usually you ask about something that is true of less than half the tuples

- If condition is $A \neq c$:
  - a good estimate is $T(R)$

- If condition is the AND of several equalities and inequalities, estimate in series.
Example

- Consider relation $R(a,b,c)$ with 10,000 tuples and 50 different values for attribute $a$.
- Consider selecting all tuples from $R$ with $a = 10$ and $b < 20$.
- Estimate of number of resulting tuples is $10,000 \times (1/50) \times (1/3) = 67$. 
If condition has the form $C_1 \text{ OR } C_2$, use:
1. sum of estimate for $C_1$ and estimate for $C_2$, or
2. minimum of $T(R)$ and the previous, or
3. assuming $C_1$ and $C_2$ are independent, $T(R) \times (1 - (1-f_1) \times (1-f_2))$, where $f_1$ is fraction of $R$ satisfying $C_1$ and $f_2$ is fraction of $R$ satisfying $C_2$
Example

- Consider relation R(a,b) with 10,000 tuples and 50 different values for a.
- Consider selecting all tuples from R with a = 10 or b < 20.
- Estimate for a = 10 is $10,000/50 = 200$
- Estimate for b < 20 is $10,000/3 = 3333$
- Estimate for combined condition is
  - $200 + 3333 = 3533$ or
  - $10,000*(1 - (1 - 1/50)*(1 - 1/3)) = 3466$
Assume join is on a single attribute Y.

Some possibilities:
1. R and S have disjoint sets of Y values, so size of join is 0
2. Y is the key of S and a foreign key of R, so size of join is $T(R)$
3. All the tuples of R and S have the same Y value, so size of join is $T(R) \times T(S)$

We need some assumptions…
Common Join Assumptions

- **Containment of Value Sets**: If R and S both have attribute Y and $V(R,Y) \leq V(S,Y)$, then every value of Y in R appears a value of Y in S
  - true if Y is a key of S and a foreign key of R

- **Preservation of Value Sets**: After the join, a non-matching attribute of R has the same number of values as it does in R
  - true if Y is a key of S and a foreign key of R
Join Estimation Rule

- Expected number of tuples in result is
  \[ T(R) \times T(S) / \max(V(R,Y), V(S,Y)) \]

- Why? Suppose \( V(R,Y) \leq V(S,Y) \).
  - There are \( T(R) \) tuples in \( R \).
  - Each of them has a \( 1/V(S,Y) \) chance of joining with a given tuple of \( S \), creating \( T(S)/V(S,Y) \) new tuples.
Example

- Suppose we have
  - $R(a,b)$ with $T(R) = 1000$ and $V(R,b) = 20$
  - $S(b,c)$ with $T(S) = 2000$, $V(S,b) = 50$, and $V(S,c) = 100$
  - $U(c,d)$ with $T(U) = 5000$ and $V(U,c) = 500$

- What is the estimated size of $R \bowtie S \bowtie U$?
  - First join $R$ and $S$ (on attribute $b$):
    - estimated size of result, $X$, is $T(R) \times T(S) / \max(V(R,b),V(S,b)) = 40,000$
    - by containment of value sets, number of values of $c$ in $X$ is the same as in $S$, namely 100
  - Then join $X$ with $U$ (on attribute $c$):
    - estimated size of result is $T(X) \times T(U) / \max(V(X,c),V(U,c)) = 400,000$
Example (cont'd)

- If the joins are done in the opposite order, still get the same estimated answer.
- Due to preservation of value sets assumption.
- This is desirable: we don't want the estimate to depend on how the result is computed.
More About Natural Join

- If there are multiple join attributes, the previous rule generalizes:
  - $T(R) \times T(S)$ divided by the larger of $V(R,y)$ and $V(S,y)$ for each join attribute $y$

- Consider the natural join of a series of relations:
  - containment and preservation of value sets assumptions ensure that the same estimated size is achieved *no matter what order the joins are done in*
Summary of Estimation Rules

- Projection: exactly computable
- Product: exactly computable
- Selection: reasonable heuristics
- Join: reasonable heuristics
- The other operators are harder to estimate…
Additional Estimation Heuristics

- **Union:**
  - bag: exactly computable (sum)
  - set: estimate as larger plus half the smaller

- **Intersection:** estimate as half the smaller

- **Difference:** estimate $R - S$ as $T(R) - T(S)/2$

- **Duplicate elimination:** $T(R)/2$ or product of all the $V(R,a)$'s, whichever is smaller

- **Grouping:** $T(R)/2$ or product of $V(R,a)$ for all grouping attributes $a$, whichever is smaller
Heuristics to Reduce Cost of LQP

- For each transformation of the tree being considered, estimate the "cost" before and after doing the transformation.
- At this point, "cost" only refers to sizes of intermediate relations (we don't yet know about number of disk I/O's).
- Sum of sizes of all intermediate relations is the heuristic: if this sum is smaller after the transformation, then incorporate it.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R(a,b)</td>
<td>S(b,c)</td>
</tr>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td></td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td></td>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>

**Initial logical query plan:**

\[ \delta \]

\[ \sigma_{a=10} \]

R \quad S
Initial logical query plan:

Modified logical query plan:
- move selection down
- should δ be moved below join?

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td></td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td></td>
<td>V(S,c) = 100</td>
</tr>
<tr>
<td>R(a,b)</td>
<td>S(b,c)</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td></td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(S,c) = 100</td>
<td></td>
</tr>
</tbody>
</table>

Initial logical query plan:

\[ \delta \]

\[ \sigma_{a=10} \]

R       S

Modified logical query plan:

- move selection down
- should \(\delta\) be moved below join?
\[ \begin{array}{|c|c|} \hline R(a,b) & S(b,c) \\ \hline T(R) = 5000 & T(S) = 2000 \\ \hline V(R,a) = 50 & \ \\ \hline V(R,b) = 100 & V(S,b) = 200 \\ \hline & V(S,c) = 100 \\ \hline \end{array} \]

Initial logical query plan:

\[ \delta \]

\[ \sigma_{a=10} \]

R \quad S

Modified logical query plan:

- move selection down
- should \( \delta \) be moved below join?

\[ \delta \quad \delta \quad vs. \quad \delta \quad \delta \]

\[ \sigma_{a=10} \quad S \]

R
Initial logical query plan:

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?

\[ \sigma_{a=10} R \quad \delta \quad S \quad vs. \quad \delta \quad \sigma_{a=10} S \]
\[
\begin{array}{|c|c|}
\hline
R(a,b) & S(b,c) \\
\hline
T(R) = 5000 & T(S) = 2000 \\
\hline
V(R,a) = 50 & \\
V(R,b) = 100 & V(S,b) = 200 \\
& V(S,c) = 100 \\
\hline
\end{array}
\]

Initial logical query plan:

- \(\delta\)

Modified logical query plan:
- move selection down
- should \(\delta\) be moved below join?

\[
\begin{array}{ccc}
\delta & \delta & \delta \\
\sigma_{a=10} & S & 2000 \\
5000 & R & \sigma_{a=10} \\
\end{array}
\]

vs.

\[
\begin{array}{ccc}
\delta & \delta & \\
\sigma_{a=10} & S & \sigma_{a=10} \\
R & & \\
\end{array}
\]
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R(a,b)</td>
<td>S(b,c)</td>
</tr>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td></td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(S,c) = 100</td>
<td></td>
</tr>
</tbody>
</table>

Initial logical query plan:

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(a,b)$</td>
<td>$S(b,c)$</td>
</tr>
<tr>
<td>T($R$) = 5000</td>
<td>T($S$) = 2000</td>
</tr>
<tr>
<td>V($R,a$) = 50</td>
<td></td>
</tr>
<tr>
<td>V($R,b$) = 100</td>
<td>V($S,b$) = 200</td>
</tr>
<tr>
<td></td>
<td>V($S,c$) = 100</td>
</tr>
</tbody>
</table>

Initial logical query plan:

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?
Initial logical query plan:

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?

VS.

$\sigma_{a=10}$

$\delta$

R

S

$\sigma_{a=10}$

S

R
### Initial logical query plan:

<table>
<thead>
<tr>
<th>$R(a,b)$</th>
<th>$S(b,c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(R) = 5000$</td>
<td>$T(S) = 2000$</td>
</tr>
<tr>
<td>$V(R,a) = 50$</td>
<td>$V(S,b) = 200$</td>
</tr>
<tr>
<td>$V(R,b) = 100$</td>
<td>$V(S,c) = 100$</td>
</tr>
</tbody>
</table>

### Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?

![Query Plan Diagram]
Table:

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(a,b)$</td>
<td>$S(b,c)$</td>
</tr>
<tr>
<td>$T(R) = 5000$</td>
<td>$T(S) = 2000$</td>
</tr>
<tr>
<td>$V(R,a) = 50$</td>
<td>$V(S,b) = 200$</td>
</tr>
<tr>
<td>$V(R,b) = 100$</td>
<td>$V(S,c) = 100$</td>
</tr>
</tbody>
</table>

Initial logical query plan:

```
\delta
```

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?

```
\sigma_{a=10} \leftarrow R, S
```

```
\delta
\sigma_{a=10} \leftarrow S, 2000
```

```
\delta
\sigma_{a=10} \leftarrow R, 5000
```

```
\delta
\sigma_{a=10} \leftarrow S, 2000
```

```
\delta
\sigma_{a=10} \leftarrow R, 5000
```

```
\delta
\sigma_{a=10} \leftarrow S, 2000
```

```
\delta
\sigma_{a=10} \leftarrow R, 5000
```

```
\delta
\sigma_{a=10} \leftarrow S, 2000
```
Initial logical query plan:

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td></td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(S,c) = 100</td>
<td></td>
</tr>
</tbody>
</table>

Modified logical query plan:
- move selection down
- should δ be moved below join?
<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td></td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td></td>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>

Initial logical query plan:

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?

Initial query plan:

Modified query plan:

vs.

vs.

$\sigma_{a=10}$
<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>

Modified logical query plan:
- move selection down
- should $\delta$ be moved below join?

Initial logical query plan:

```
\[
\sigma_{a=10} \left( \delta_{\leq 10} (R \bowtie S) \right)
\]
```

```
\[
\delta_{\leq 10} \left( \sigma_{a=10} (R \bowtie S) \right)
\]
```
<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>2000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V(R,a) = 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>V(R,b) = 100</td>
</tr>
<tr>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>

**Initial logical query plan:**

- δ
- σ_{a=10}

**Modified logical query plan:**
- move selection down
- should δ be moved below join?

![Diagram showing query plans with numbers and operators]
Initial logical query plan:

Modified logical query plan:
- move selection down
- should \( \delta \) be moved below join?

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>S(b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 5000</td>
<td>T(S) = 2000</td>
</tr>
<tr>
<td>V(R,a) = 50</td>
<td>V(S,b) = 200</td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,c) = 100</td>
</tr>
</tbody>
</table>
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Deriving a Physical Query Plan

To convert a logical query plan into a physical query plan, choose:
- an order and grouping for sets of joins, unions, and intersections
- algorithm for each operator (e.g., nest-loop join vs. hash join)
- additional operators (scanning, sorting, etc.) that are needed for physical plan but not explicitly in the logical plan
- how to pass arguments (store intermediate result on disk vs. pipeline one tuple or buffer at time)

Physical query plans are evaluated by their estimated cost…
Cost of Evaluating an Expression

- Measure by number of disk I/O's
- Influenced by:
  - operators in the chosen logical query plan
  - sizes of intermediate results
  - physical operators used to implement the logical operators
  - ordering of groups of similar operators (e.g., joins)
  - argument passing method
Some Heuristics

- To implement selection on R with condition $A = c$: if R has an index on A, then use index-scan
- To implement join when one argument R has an index on the join attribute(s): use index-join with R in inner loop
- To implement join when one argument R is sorted on the join attribute(s): choose sort-join over hash-join
- To implement union or intersection of > 2 relations: group smallest relations first
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Choosing Order for Joins

- Suppose we have > 2 relations to be joined (naturally)
- Pay attention to asymmetry:
  - one-pass alg: left argument is smaller and is stored in main memory data structure
  - nested-loop alg: left argument is used in the outer loop
  - index-join: right argument has the index
- Common point: these algs work better if left argument is the smaller one
Choosing Join Order (cont'd)

- Template for tree is given below:
- Choices are which relations go where:
Outline

- Convert SQL query to a parse tree
  - Semantic checking: attributes, relation names, types
- Convert to a logical query plan (relational algebra expression)
  - deal with subqueries
- Improve the logical query plan
  - use algebraic transformations
  - group together certain operators
  - evaluate logical plan based on estimated size of relations
- Convert to a physical query plan
  - search the space of physical plans
  - choose order of operations
  - complete the physical query plan
Remaining Steps

- Choose algorithms for remaining operators
- Decide when intermediate results will be **materialized** (stored on disk in entirety) or **pipelined** (created only in main memory, in pieces)
Materialization vs. Pipelining

- **Materialization**: perform operations in series and write intermediate results to disk
- **Pipelining**: interleave execution of several operations. Tuples produced by one operation are passed directly to the operations that use them as input, bypassing the disk
  - saves on disk I/O's
  - requires more main memory
Example Physical Query Plans

Filter\( (x=1 \text{ AND } z<5) \)

IndexScan\( (R, y=2) \)

\[ \sigma_{x=1 \text{ AND } y=2 \text{ AND } z<5} (R) \]

TableScan\( (R) \)

TableScan\( (S) \)

TableScan\( (U) \)

two-pass hash-join
101 buffers

materialize

two-pass hash-join
101 buffers

R \bowtie S \bowtie U