Query processing

• Understanding query processing helps producing better applications

• SQL is a declarative language: it describes the query result, but not how to get it.

• Query processing:
  – Query analysis → logical query plan
  – Query transformation
  – Physical plan generation and optimization
  – Query execution
Physical db design

• A query optimizer uses all available indexes, materialized views, etc. in order to better execute the query
  – Data Base Administrator (DBA) is expected to set up a good physical design
  – Good DBAs understand query optimizers very well
  – Good DBAs are hard to find
Query execution steps: analysis

SQL COMMAND

CATALOG

ANALYSIS AND SIMPLIFICATION

LOGICAL OPERATOR TREE

QUERY TRANSFORMATION

SELECT Name
FROM Students S, Exams E
WHERE S.StudCode = E.Candidate AND City = 'PI' AND Grade > 25

Check command rewrite Boolean conditions produce logical tree

\[ \pi_{\text{Name}} \]
\[ \sigma_{\text{City} = 'PI' \text{ and Grade} > 25} \]
\[ S.\text{StudCode} = E.\text{Candidate} \]

Students S

Exams E
Query execution steps: transformation

Transform a logical query plan using equivalence rules to get a faster plan.

```
π_{Name}
/
σ_{City = 'PI' and Grade > 25}
/

S.StudCode = E.Candidate
```

```
π_{Name}
/

S.StudCode = E.Candidate
/

σ_{City = 'PI'}
/

σ_{Grade > 25}
/

Students S
/

Exams E
```
Select an algorithm for each logical operation.

**Ideally**: Want to find **best** physical plan.

**In practice**: Avoid **worst** physical plans!

---

**Query ex. steps: physical plan generation**

---

**PHYSICAL PLAN GENERATION**

**PHYSICAL PLAN:**

**PHYSICAL OPERATORS**

**TREE**

**PLAN EXECUTION**

**RESULT**

---

**Project**

$$\{\text{Name}\}$$

**NestedLoop**

$$\text{(S.StudCode=E.Candidate)}$$

**IndexFilter**

$$\text{(Students,IdxP, City = 'Pi') }$$

**Filter**

$$\text{(Grade > 25) }$$

**TableScan**

$$\text{(Exams)}$$
Physical plan execution

• Each operator is implemented as an iterator using a ‘pull’ interface: when an operator is ‘pulled’ for the next output tuples, it ‘pulls’ on its inputs and computes them.

• An operator interface provides the methods open, next, isDone, and close implemented using the Storage Engine interface.
Interesting transformations

• **DISTINCT** Elimination
• **GROUP BY** Elimination
• **WHERE**-Subquery Elimination
• **VIEW** Elimination (Merging)
• Many are based on functional dependencies
• Do you remember functional dependencies?
Functional dependencies

• For $R(T)$ and $X, Y \subseteq T$

• $X \rightarrow Y$ (X determines Y) iff:
  
  - $\forall r$ valid instance of $R$.
  
  $\forall t_1, t_2 \in r$. If $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$
## Example

<table>
<thead>
<tr>
<th>StudCode</th>
<th>Name</th>
<th>City</th>
<th>Region</th>
<th>BirthYear</th>
<th>Subject</th>
<th>Grade</th>
<th>Univ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>Mary</td>
<td>Pisa</td>
<td>Tuscany</td>
<td>1995</td>
<td>DB</td>
<td>30</td>
<td>Pisa</td>
</tr>
<tr>
<td>1234567</td>
<td>Mary</td>
<td>Pisa</td>
<td>Tuscany</td>
<td>1995</td>
<td>SE</td>
<td>28</td>
<td>Pisa</td>
</tr>
<tr>
<td>1234568</td>
<td>John</td>
<td>Lucca</td>
<td>Tuscany</td>
<td>1994</td>
<td>DB</td>
<td>30</td>
<td>Pisa</td>
</tr>
<tr>
<td>1234568</td>
<td>John</td>
<td>Lucca</td>
<td>Tuscany</td>
<td>1994</td>
<td>SE</td>
<td>28</td>
<td>Pisa</td>
</tr>
</tbody>
</table>

- StudCode $\rightarrow$ Name, City, Region, BirthYear
- City $\rightarrow$ Region
- StudCode, Subject $\rightarrow$ Grade
- $\emptyset$ $\rightarrow$ Univ
- StudCode, Name $\rightarrow$ City, Univ, Name
Functional dependencies

- Trivial dependencies: $XY \rightarrow X$
- Atomic dependency: $X \rightarrow A$ (A attribute)
- Union rule:
  - $X \rightarrow A_1...A_n$ iff $X \rightarrow A_1$ ... $X \rightarrow A_n$
- What about the lhs:
  - Does $A_1...A_n \rightarrow X$ imply $A_1 \rightarrow X$ ... $A_n \rightarrow X$?
  - Does $A_1 \rightarrow X$ imply $A_1..A_n \rightarrow X$?
- What does $\emptyset \rightarrow X$ mean?
Functional dependencies and keys

• Canonical dependencies:
  – $X \rightarrow A$ but not $X' \rightarrow A$, for any $X' \subseteq X$

• Every non-trivial dependency ‘contains’ one or more canonical dependencies – just remove extraneous attributes

• Key: set $K$ such that $K \rightarrow T$ holds and is canonic

• In a well designed relation, only one kind of non-trivial canonical dependencies (BCNF):
  – Key $\rightarrow A$ (key dependencies)
Deriving dependencies

• Given a set F of FDs, \( X \rightarrow Y \) is derivable from F (\( F \models X \rightarrow Y \)), iff \( X \rightarrow Y \) can be derived from F using the following rules:
  
  – If \( Y \subseteq X \), then \( X \rightarrow Y \) (Reflexivity R )
  
  – If \( X \rightarrow Y \) and \( Z \subseteq T \), then \( XZ \rightarrow YZ \) (Augmentation A)
  
  – If \( X \rightarrow Y \) and \( Y \rightarrow Z \), then \( X \rightarrow Z \) (Transitivity T)

• Soundness:
  
  – when \( r \models F \) and \( F \models X \rightarrow Y \), then \( r \models X \rightarrow Y \)
Closure of an attribute set

• **Definition** Given \( R<T, F> \), and \( X \subseteq T \), the *closure* of \( X \) wrt \( F \), denoted by \( X_F^+ \), (or just \( X^+ \) when \( F \) is clear), is:

\[
- X_F^+ = \{ A_i \in T \mid F \vdash X \rightarrow A_i \}
\]

• **Theorem:** \( F \vdash X \rightarrow Y \iff Y \subseteq X_F^+ \)
Example

• StudCode → Name, City, BirthYear
• City → Region
• StudCode, Subject → Grade
• Ø → Univ

• StudCode⁺ = {StudCode, Name, City, BirthYear, Region, Univ}
• (StudCode, Name)⁺ = {
• (Name,City)⁺ = {Name, City, Region, Univ}
• (StudCode, Subject)⁺
• Ø⁺
Dependencies in a SQL query

• Consider a query on a set of tables \( R_1(T_1), \ldots, R_n(T_n) \) such that no attribute name appears in two tables.

• After joins and select, assuming that the WHERE condition \( C \) is in CNF, these dependencies hold on the result:
  
  – The initial dependencies: \( K_{ij} \rightarrow T_i \) for any key \( K_{ij} \) of the table \( T_i \)
  
  – Constant dependencies \( \emptyset \rightarrow A \) for any factor \( A=c \) in \( C \)
  
  – Join dependencies \( A_i \rightarrow A_j \) and \( A_j \rightarrow A_i \) for any factor \( A_i=A_j \)
Computing the closure of X

• Assume a product-select-project expression with CNF condition

• Let $X^+ = X$

• Add to $X^+$ all attributes $A_i$ such that $A_i = c$ is in C

• Repeat until $X^+$ stops changing:
  – Add to $X^+$ all $A_j$ such that $A_k$ is in $X^+$ and $A_j = A_k$ or $A_k = A_j$ is in C
  – Add to $X^+$ all attributes of $R_i$ if one key of $R_i$ is included in $X^+$
DISTINCT elimination

• Consider a SELECT DISTINCT query
  – Duplicate elimination is very expensive, and DISTINCT is often redundant

• SELECT Name FROM Students
• SELECT StudId FROM Students
• SELECT StudId FROM Students NATURAL JOIN Exams
DISTINCT elimination

• Consider E returning a set of tuples of type \{T\}. If $A \rightarrow T$, then $\pi^b_A(E)$ creates no duplicates: if two lines coincide on $A$ they are the same line

• SELECT DISTINCT A
FROM R1(T1),...,Rn(Tn)
WHERE C:
  – DISTINCT is redundant when $A^+$ is $T1 \cup ... \cup Tn$ (or $A^+$ includes a key for every relation in the join), assuming that all input tables are sets (have a key)
  – $A^+$ can be computed as in the previous slide
Distinct elimination: example

Products(PkProduct, ProductName, UnitPrice)
Invoices(PkInvoiceNo, Customer, Date, TotalPrice)
InvoiceLines(FkInvoiceNo, LineNo, FkProduct, Qty, Price)

SELECT DISTINCT FkInvoiceNo, TotalPrice
FROM InvoiceLines, Invoices
WHERE FkInvoiceNo = PkInvoiceNo;

SELECT DISTINCT FkInvoiceNo, TotalPrice
FROM InvoiceLines, Invoices
WHERE FkInvoiceNo = PkInvoiceNo AND LineNo = 1;
DISTINCT elimination with GROUP BY

• Consider a GROUP BY query:
  – SELECT DISTINCT X, f
  – FROM R1,...,Rn WHERE C1
  – GROUP BY X,Y   HAVING C2

• The set X,Y determines all other attributes in the output of the run-time \( \{X,Y\} \gamma \{f,g\} \) operation

• Hence, DISTINCT is redundant when \( XY \subseteq X+ \)
• The X+ computation has to use the keys of R1,...,Rn and the conditions C1 and C2
Distinct elimination: example

SELECT DISTINCT FkInvoiceNo, COUNT(*) AS N
FROM InvoiceLines, Invoices
WHERE FkInvoiceNo = PkInvoiceNo
GROUP BY FkInvoiceNo, Customer;
Group by elimination

Products(PkProduct, ProductName, UnitPrice)
Invoices(PkInvoiceNo, Customer, Date, TotalPrice)
InvoiceLines(FkInvoiceNo, LineNo, FkProduct, Qty, Price)

SELECT       FkInvoiceNo, COUNT(*) AS N
FROM          InvoiceLines, Invoices
WHERE         FkInvoiceNo = PkInvoiceNo
              AND TotalPrice > 10000  AND  LineNo = 1
GROUP BY      FkInvoiceNo, Customer;

The query producing the data to be grouped is without duplicates?

SELECT       FkInvoiceNo, Customer
FROM          InvoiceLines, Invoices
WHERE         FkInvoiceNo = PkInvoiceNo
              AND TotalPrice > 10000  AND  LineNo = 1;
WHERE-subquery elimination

```
select *                   nested correlated
from students s
where exists (select * from exams e where e.sid=s.sid)
```

```
select *                   nested not correlated
from students s
where s.id in (select e.sid from exams e)
```

```
select distinct s.*        unnested
from students s natural join exams e
```
WHERE-subquery elimination

• The most important transformation: very common and extremely relevant
• Very difficult problem: no general algorithm
• We only consider here the basic case:
  – Subquery is EXISTS (do not consider NOT EXISTS)
  – Correlated subquery
  – Subquery with no GROUP BY
# Left outer join

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**SQL:**

```
SELECT * FROM R
NATURAL LEFT JOIN S;
```

Also called: natural left outer join

---

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

**SQL:**

```
SELECT * FROM R
NATURAL JOIN S;
```

Also called: natural inner join

---

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

**SQL:**

```
SELECT * FROM R
NATURAL LEFT JOIN S;
```

Also called: natural left outer join
## Outer join: right, full

### SELECT * FROM R NATURAL RIGHT JOIN S;

Also called: natural right **outer** join

### SELECT * FROM R NATURAL FULL JOIN S;

Also called: natural full **outer** join
WHERE unnesting

• Courses(CrsName, CrsYear, Teacher, Credits)
• Transcripts(StudId, CrsName*, Year, Date, Grade)
WHERE unnesting

SELECT *
FROM Courses C
WHERE CrsYear = 2012 AND EXISTS (SELECT FROM Transcripts T
    WHERE T.CrsName = C.CrsName AND T.Year = CrsYear);

• The unnested equivalent query is

SELECT DISTINCT C.*
FROM Courses C, Transcripts T
WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
AND CrsYear = 2012;
WHERE unnesting

SELECT DISTINCT C.Teacher
FROM Courses C
WHERE CrsYear = 2012 AND
EXISTS (SELECT FROM Transcripts T
      WHERE T.CrsName = C.CrsName
      AND T.Year = CrsYear);

• The unnested equivalent query is

SELECT DISTINCT C.Teacher
FROM Courses C, Transcripts T
WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
AND CrsYear = 2012;
WHERE unnesting

SELECT C.Teacher
FROM Courses C
WHERE CrsYear = 2012 AND
EXISTS (SELECT FROM Transcripts T
    WHERE T.CrsName = C.CrsName
    AND T.Year = CrsYear);

• Is not equivalente to the following, w or w/o distinct:

SELECT (DISTINCT) C.Teacher
FROM Courses C, Transcripts T
WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
AND CrsYear = 2012;
WHERE unnesting

• SELECT C.CrsName, C.Teacher
  FROM Courses C
  WHERE CrsYear = 2012 AND
    EXISTS ( SELECT count(*) FROM Transcripts T
      WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
      HAVING 27 < AVG(Grade))

• The unnested equivalent query is

• SELECT C.CrsName, C.Teacher
  FROM Courses C, Transcripts T
  WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
    AND CrsYear = 2012
  GROUP BY C.CrsName, C.Teacher
  HAVING 27 < AVG(Grade);
WHERE unnesting

• SELECT C.CrsName, C.Teacher
  FROM Courses C
  WHERE C.CrsYear = 2012 AND
    EXISTS ( SELECT count(*) FROM Transcripts T
      WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
      HAVING 0 = Count(*) )

• The following is wrong (the count bug problem)
• SELECT C.CrsName, C.Teacher
  FROM Courses C, Transcripts T
  WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
    AND CrsYear = 2012
  GROUP BY C.CrsName, C.Teacher
  HAVING 0 = Count(*) ;
WHERE unnesting

- **SELECT** C.CrsName, C.Teacher  
  **FROM** Courses C  
  **WHERE** C.CrsYear = 2012 **AND**  
  **EXISTS** ( **SELECT** * FROM Transcripts T  
    **WHERE** T.CrsName = C.CrsName **AND** T.Year = CrsYear  
    **HAVING** 0 = Count(*))

- The following is ok:

- **SELECT** C.CrsName, C.Teacher  
  **FROM** Courses C **LEFT JOIN** Transcripts T  
  **ON** (T.CrsName = C.CrsName **AND** T.Year = CrsYear)  
  **WHERE** CrsYear = 2012  
  **GROUP BY** C.CrsName, C.Teacher  
  **HAVING** 0 = Count(C.Grade);
View merging

- CREATE VIEW TestView AS
  SELECT Price, AName
  FROM Order, Agent
  WHERE FKAgent = PKAgent;
- SELECT Price, AName
  FROM TestView
  WHERE Price = 1000;
- Can the query be transformed to avoid the use of the view?
Temporary view

• Created by a SELECT in the FROM:

• SELECT ...
  FROM  (SELECT ... FROM ...) AS Q1,
  (SELECT ... FROM ...) AS Q2,
  WHERE ...

• Same as

• WITH  Q1 AS (SELECT ... FROM ...)
  , Q2 AS (SELECT ... FROM ...)
SELECT .... FROM  Q1, Q2, WHERE ...

View merging

The approach:

Let a Logical Plan be

View merging:

(1) View Logical Plan

(2) Query Logical Plan

(3) Query View

(4) Query Transformed Logical Plan

π

σ_H

γ

σ_W

SQL without View
View merging: an equivalence rule

• Let $X_R$ be attributes of $R$ with $f_k \in X_R$ a foreign key of $R$ referring to $pk$ of $S$ with attributes $A(S)$, then

$$(X_R \gamma F(R)) \mathrel{\bowtie} f_k = p_k S \equiv X_R \cup A(S) \gamma F(R) \mathrel{\bowtie} f_k = p_k S'$$
CREATE VIEW TestView AS
SELECT Price, AName
FROM Order, Agent
WHERE FKAgent = PKAgent;

SELECT Price, AName
FROM TestView
WHERE Price = 1000;

TestView =
\[ \pi_{Price, AName} \]
(1)
FKAgent = PKAgent
\[ \bowtie \]
Order \quad Agent

\[ \pi_{Price, AName} \]
(2)
\[ \sigma_{Price = 1000} \]
TestView

\[ \pi_{Price, AName} \]
(3)
\[ \sigma_{Price = 1000} \]
\[ \pi_{Price, AName} \]
FKAgent = PKAgent
\[ \bowtie \]
Order \quad Agent

\[ \pi_{Price, AName} \]
(4)
\[ \sigma_{Price = 1000} \]
FKAgent = PKAgent
\[ \bowtie \]
Order \quad Agent
CREATE VIEW FKAgent_GBY AS
SELECT FKAgent, COUNT(*) AS No
FROM Order
GROUP BY FKAgent;

SELECT AName, No
FROM FKAgent_GBY, Agent
WHERE FKAgent = PKAgent
AND ACity = 'Pisa';
CREATE VIEW FKAgent_GBY AS
SELECT FKAgent, COUNT(*) AS No
FROM Order
GROUP BY FKAgent;

SELECT AName, No
FROM FKAgent_GBY, Agent
WHERE FKAgent = PKAgent
AND ACity = 'Pisa';

GROUP BY FKAgent, AName;

\[
\left( X_R \gamma F(R) \right) \underset{f_k=p_k}{\bowtie} S \equiv X_R \cup A(S) \gamma F(R) \underset{f_k=p_k}{\bowtie} S
\]
Physical plan generation

• Main steps:
  – Generate plans
  – Evaluate their cost

• Plan generation:
  – Needs to keep track of attributes and order of each intermediate result

• Cost evaluation:
  – Evaluate the size of each intermediate result
  – Evaluate the cost of each operator
Physical plan generation phase: statistics and catalog

- The Catalog contains the following statistics:
  - \( N_{\text{reg}} \) and \( N_{\text{pag}} \) for each relation.
  - \( N_{\text{key}} \) and \( N_{\text{leaf}} \) for each index.
  - min/max values for each index key.
  - ... Histograms

- The Catalog is updated with the command \textbf{UPDATE STATISTICS}
Single relation queries

• \( S(PkS, FkR, aS, bS, cS) \)
• SELECT \( bS \)
  FROM \( S \)
  WHERE \( FkR > 100 \) AND \( cS = 2000 \)
• The only question is which index or indexes to use
• If we have an index on \( (cS, FkR, bS) \), an IndexOnly plan can be used
Multiple relation queries

• Basic issue: join order
• Every permutation is a different plan
  – AxBxCxD
  – BxAxCxD
  – BxCxAxD
  – ...
• $n!$ permutations
Multiple relation queries

• Every permutation is many different plans
  – $Ax(Bx(CxD))$
  – $(AxB)x(CxD)$
  – $(Ax(BxC))xD$
  – $Ax((BxC)xD)$
  – ...

• Many different choices of join operator

• Huge search space!
Full search

R Join S Join T

One relation
S1 \rightarrow S2 \rightarrow S3

Two relations
S4 \rightarrow S5 \rightarrow S10

Three relations
S6

= Physical plan min cost
Optimization algorithm for a join

• Initialize Plans with one tree for each restricted relation
• repeat {
  extract from Plans the fastest plan P
  if P is complete, exit.
  else, expand P:
    join P with all other plans $P'$ on disjoint relations
    for each $P$ join $P'$, put the best tree in Plans
    remove $P$
}
Optimization algorithm: heuristics

- Left deep: generate left-deep trees only
- Greedy: after a node is expanded, only expand its expansions
- Iterative full search: alternate full and greedy
- Interesting-order plans should also be considered
Example

R(N, D, T, C), with indexes on C and T
S(C, O, E), with indexes on C and E

SELECT S.C, S.O
FROM S, R
WHERE S.C = R.C AND E = 13 AND T = ‘AA’;

\[ \pi_{S.C, S.O}^{b}(\sigma_{E = 13 \land T = ‘AA’}(S \bowtie R)) \]

\[ \pi_{S.C, S.O}^{b}(\text{\underline{\sigma}_{E = 13}(S) \bowtie \sigma_{T = ‘AA’}(R)}) \]
Example

R(N, D, T, C), with indexes on C and T

S(C, O, E), with indexes on C and E

\[ \pi_{S \cdot C, S \cdot O}^{b} \left( \sigma_E = 13(S) \bowtie \sigma_T = 'AA'(R) \right) \]

Physical plans for subexpression on relations

minimum cost
Example

\[ \pi^b_{S.C, S.O}(\sigma_{E=13} (S) \bowtie \sigma_{T='AA'} (R)) \]

\[
\begin{align*}
\sigma_{E=13} (S) & \quad \bowtie \quad \sigma_{T='AA'} (R) \\
\text{IndexNestedLoop} & \quad \text{(S.C = R.C)} \\
\text{NestedLoop} & \quad \text{(S.C = R.C)} \\
\text{IndexFilter} & \quad \text{(S, IdxE, E=13)} \\
\text{IndexFilter} & \quad \text{(R, IdxT, T='AA')} \\
\text{Filter} & \quad \text{(T='AA')} \\
\text{IndexFilter} & \quad \text{(R, IdxRC, C=S.C)} \\
\end{align*}
\]

\[
\begin{align*}
\sigma_{T='AA'} (R) & \quad \bowtie \quad \sigma_{E=13} (S) \\
\text{IndexNestedLoop} & \quad \text{(R.C = S.C)} \\
\text{NestedLoop} & \quad \text{(R.C = S.C)} \\
\text{IndexFilter} & \quad \text{(R, IdxT, T='AA')} \\
\text{IndexFilter} & \quad \text{(R, IdxRC, C=S.C)} \\
\text{Filter} & \quad \text{(E=13)} \\
\end{align*}
\]

\[ \times \quad \times \quad \text{minimum cost} \]
Example

\[ \pi_{S.C, S.O}^{b}(\sigma_{E = 13}(S) \bowtie \sigma_{T = 'AA'}(R)) \]

Final physical plan
Optimization of queries with grouping and aggregations

• The standard way to evaluate queries with group-by is to produce a plan for the join, and then add the group-by

• To produce cheaper physical plans the optimizer should consider doing the group-by before the join
Example

```
SELECT FKAgent, SUM(Qty) AS SQ
FROM Order, Agent
WHERE FKAgent = PKAgent AND ACity = 'Pisa'
GROUP BY FKAgent;
```
Pre-grouping

Standard Physical Plan

Physical Plan with the Pre-Grouping

HashGroupBy
({FKAgent}, {SUM(Qty) AS SQ})

NestedLoop
(PKAgent = FKAgent)

Filter
(ACity = 'Pisa')

TableScan
(Order)

TableScan
(Agent)

Project
({FKAgent, SQ})

NestedLoop
(FKAgent = PKAgent)

Filter
(ACity = 'Pisa')

TableScan
(Agent)

TableScan
(Order)

HashGroupBy
({FKAgent}, {SUM(Qty) AS SQ})

TableScan
(Agent)

TableScan
(Order)

SELECT FKAgent, SUM(Qty) AS SQ
FROM Order, Agent
WHERE FKAgent = PKAgent and ACity = 'Pisa'
GROUP BY FKAgent;
Assumptions

• The tables do not have null values, and primary and foreign keys have only one attribute
• The queries are a single SELECT with GROUP BY and HAVING but without subselect, DISTINCT and ORDER BY clauses
• In the SELECT there are all the grouping attributes
The pre-grouping problem

\[ x \gamma_F (R \times_{f_k=p_k} S) \]

When and how can the group-by be pushed through the join?

\[ x \gamma_F (R \times_{f_k=p_k} S) \equiv \ldots ((x', \gamma_{F'}(R)) \times_{f_k=p_k} S) \]
Grouping equivalence rules: $\sigma$

$$\sigma_{\phi}(x\gamma_f(E)) \equiv x\gamma_f(\sigma_{\phi}(E))$$

Two cases to consider for the selection

1) $\sigma_{\phi_X}(x\gamma_f(E)) \equiv x\gamma_f(\sigma_{\phi_X}(E))$  \hspace{1cm} \text{In SQL}

2) $\sigma_{\phi_F}(x\gamma_{\text{AGG}(A_1)} \text{ AS } F_1, \ldots, \text{AGG}(A_n) \text{ AS } F_n(E))$

\hspace{1cm} $\text{AGG} = \text{COUNT, SUM, MIN, MAX, AVG}$

Bad news: two cases only

$$\sigma_{Mb \geq \nu(x\gamma_{\text{MAX}(b)} \text{ AS } Mb(E))} \equiv x\gamma_{\text{MAX}(b)} \text{ AS } Mb(\sigma_{b \geq \nu(E)})$$

$$\sigma_{mb \leq \nu(x\gamma_{\text{MIN}(b)} \text{ AS } mb(E))} \equiv x\gamma_{\text{MIN}(b)} \text{ AS } mb(\sigma_{b \leq \nu(E)})$$
Grouping equivalence rules

Assume that $X \rightarrow Y$:

$$X \gamma_F(E) \equiv \pi^b_{X \cup F}(X \cup Y \gamma_F(E))$$

<table>
<thead>
<tr>
<th>PKOrder</th>
<th>FKAgent</th>
<th>...</th>
<th>PKAgent</th>
<th>AName</th>
<th>ACity</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>Rossi</td>
<td>Pisa</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>...</td>
<td>2</td>
<td>Verdi</td>
<td>Firenze</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>Rossi</td>
<td>Pisa</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>...</td>
<td>2</td>
<td>Verdi</td>
<td>Firenze</td>
<td>...</td>
</tr>
</tbody>
</table>
Grouping equivalence rules

• Let $F$ be decomposable with $F_l - F_g$

\[ x \gamma_F(E) \equiv x \gamma_{F_g}(x \cup Y \gamma_{F_l}(E)) \]
The pre-grouping problem

\[ X \gamma_F(R_{f_k=p_k} \ S) \]

When and how can the group-by be pushed through the join?

\[ X \gamma_F(R_{f_k=p_k} \ S) \equiv \ldots ((X, \gamma_{F'})((R))_{f_k=p_k} \ S) \]

Three cases
The invariant grouping rule

**Proposition 1.** $R$ has the **invariant grouping** property

\[
X \gamma_F(R \bowtie^C_j S) \equiv \pi^b_{X \cup F}((X \cup A(C_j) - A(S) \gamma_F(R)) \bowtie^C_j S)
\]

if the following conditions are true:

1. $C_j |\rightarrow X \rightarrow A(S)$: in every group, only one line from $S$
   - in practice: $C_j$ is $f_k = p_k$, with $f_k$ in $R$, $p_k$ key for $S$, $X \rightarrow f_k$

2. Each aggregate function in $F$ only uses attributes from $R$. 
Example

```
SELECT PKAgent, SUM(Qty) AS SQ
FROM Order, Agent
WHERE FKAgent = PKAgent AND ACity = 'Pisa'
GROUP BY PKAgent;
```
Example

\[ x \gamma_F(R_{C_j} S) \equiv \]

\[ \pi^b_{X \cup F}((X \cup A(C_j) - A(S) \gamma_F(R))_{C_j} S) \]

\[ \pi^b_{PKAgent, SQty} \]

\[ \times \]

\[ \pi^b_{FKAgent=PKAgent} \]

\[ FKAgent \gamma_{SUM(Qty)} AS SQty \]

\[ \sigma_{ACity = 'Pisa'} \]

\[ Order \]

\[ Agent \]
Tests

SELECT PKAgent, ACity, SUM(Qty) AS SQ 
FROM Order, Agent 
WHERE FKAgent = PKAgent 
GROUP BY PKAgent, ACity;

SELECT ACity, SUM(Qty) AS SQ 
FROM Order, Agent 
WHERE FKAgent = PKAgent 
GROUP BY ACity;

SELECT AName, SUM(Qty) AS SQ 
FROM Order, Agent 
WHERE FKAgent = PKAgent AND ACity = 'Pisa' 
GROUP BY AName;
Summary

- Understand principles and methods of query processing in order to produce a good physical design and better applications
- Query rewriting
- Production of alternative plans and cost evaluation