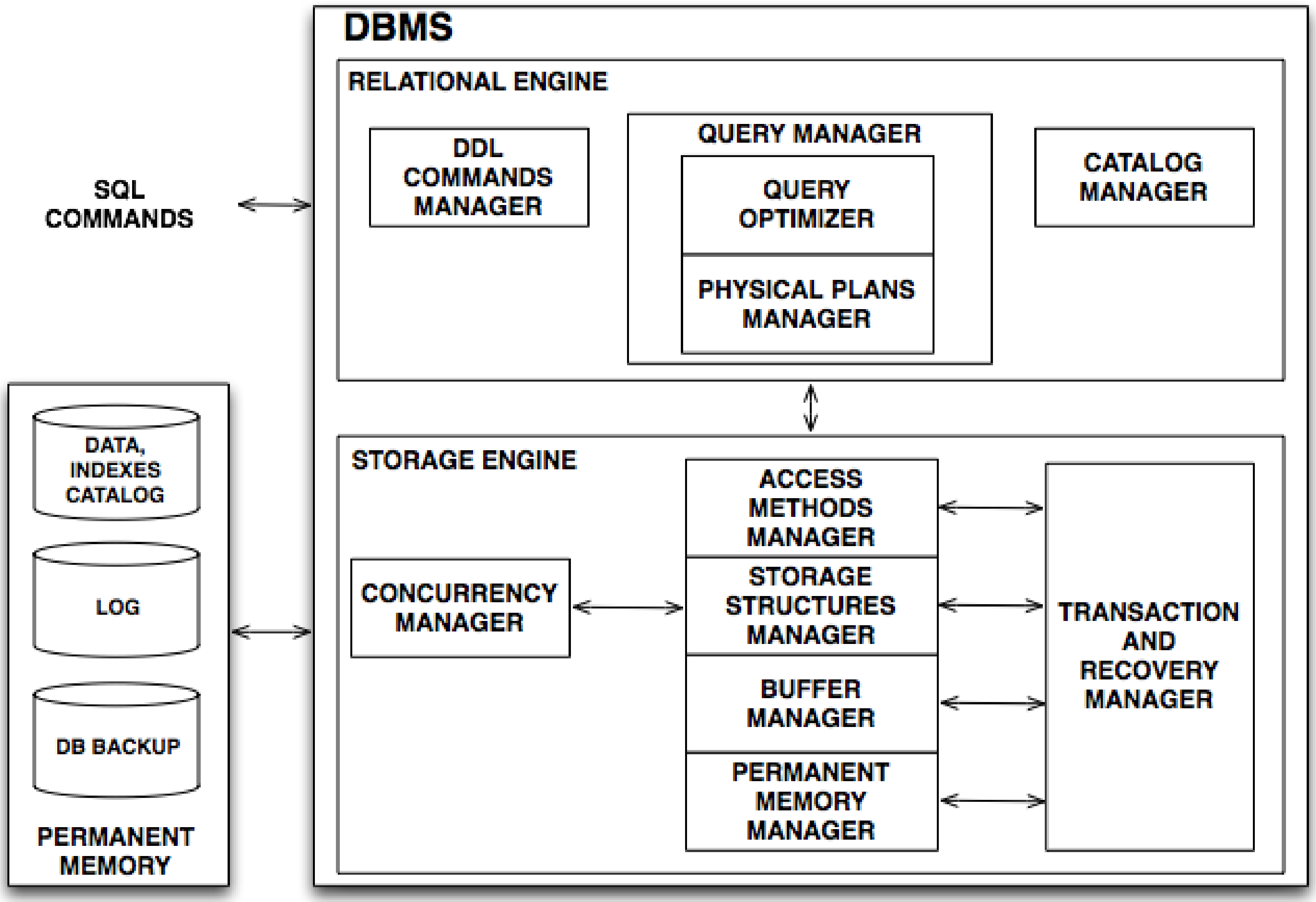
Architecture



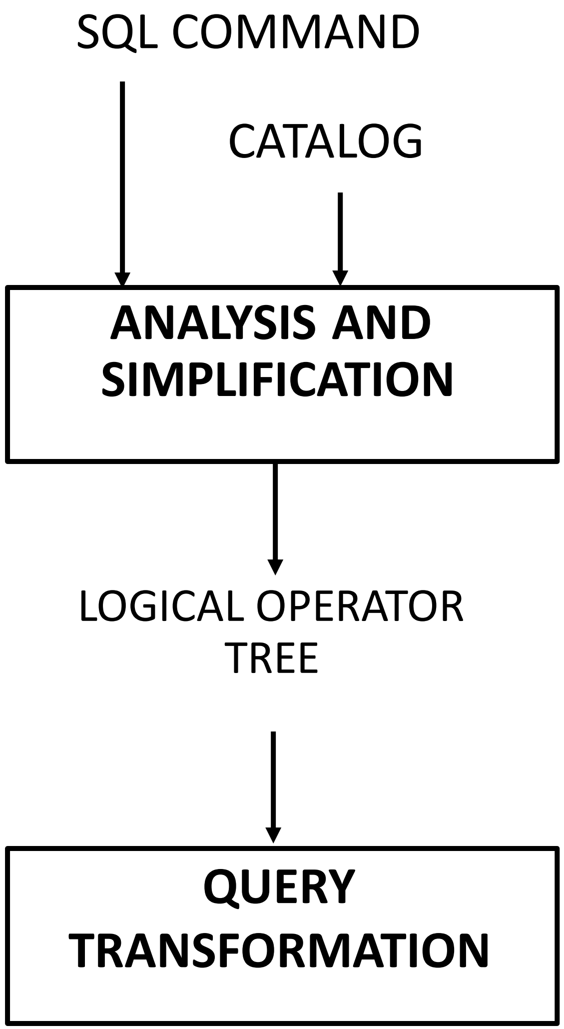
Query processing

- 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 $\rightarrow$ logical query plan
  - Query transformation
  - Physical plan generation and optimization
  - Query execution

Physical db design

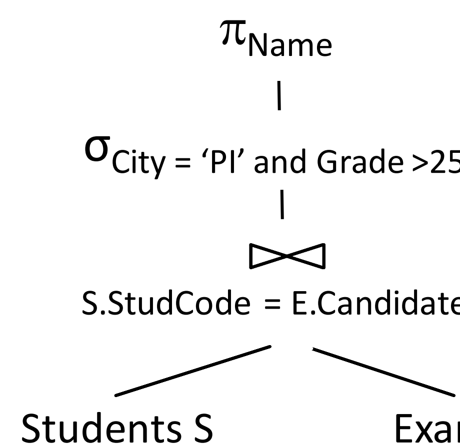
- A query optimizer uses all available indexes, materialized views, etc. in order to better execute the query
  - Data Base Administrator (DBA) is expected to set up a good physical design
  - Good DBAs understand query optimizers very well
  - Good DBAs are hard to find

Query execution steps: analysis

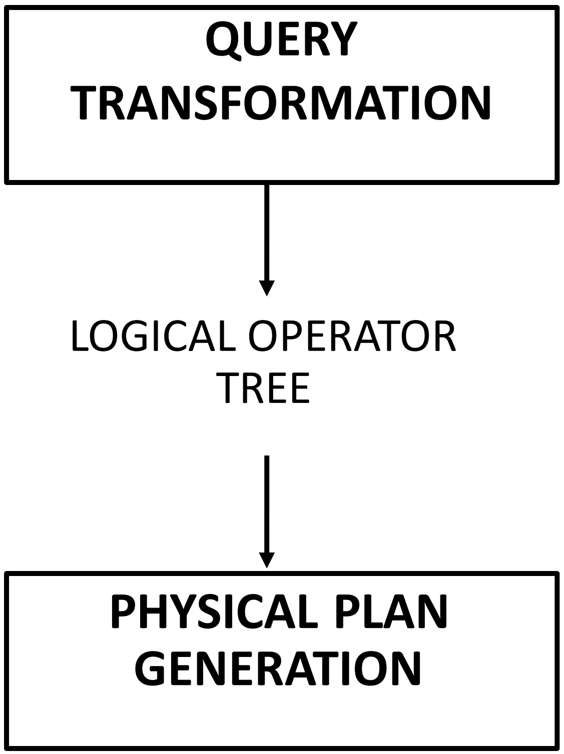



```
SELECT  Name
FROM    Students S, Exams E
WHERE   S.StudCode = E.Candidate AND
        City='PI' AND Grade>25
```

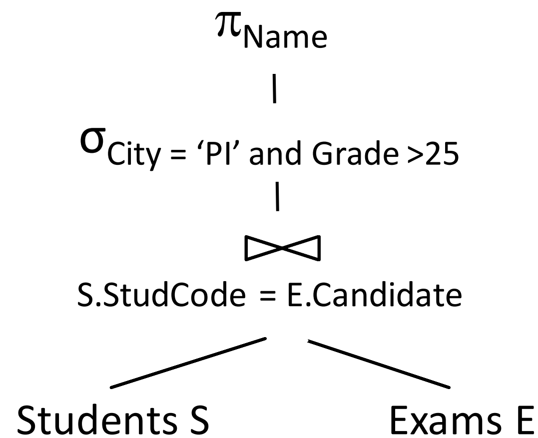
Check command
rewrite Boolean conditions
produce logical tree

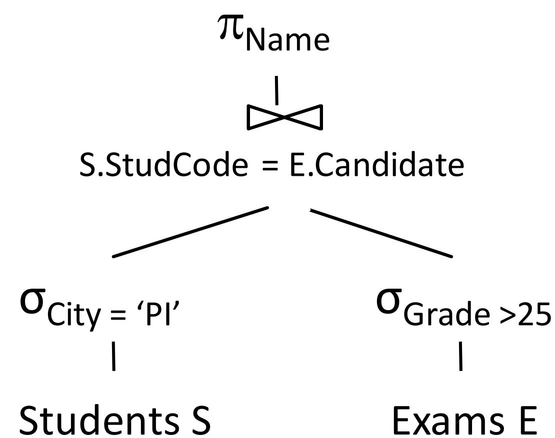


Query execution steps: transformation

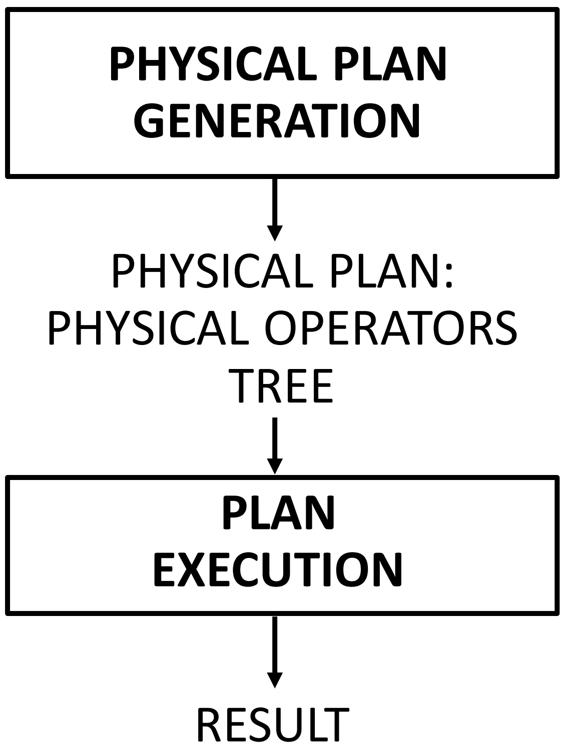


Transform a logical query plan using equivalence rules to get a faster plan





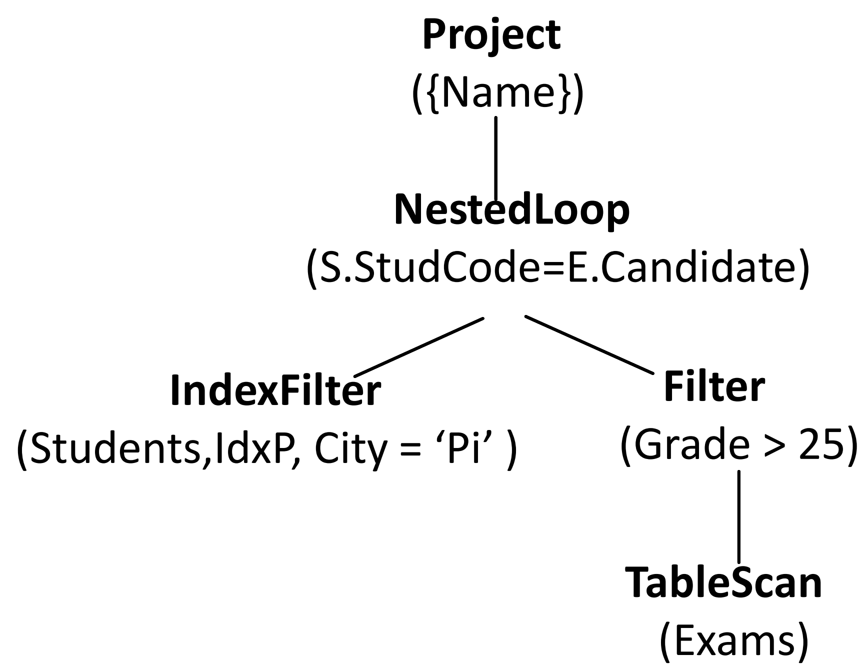
Query ex. steps: physical plan generation



Select an algorithm for each logical operation.

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

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



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.
    $\forall$ t1, t2 $\in$ r. If t1[X] = t2[X] then t1[Y] = t2[Y]

Example

StudCode	Name	City	Region	BirthYear	Subject	Grade	Univ
1234567	Mary	Pisa	Tuscany	1995	DB	30	Pisa
1234567	Mary	Pisa	Tuscany	1995	SE	28	Pisa
1234568	John	Lucca	Tuscany	1994	DB	30	Pisa
1234568	John	Lucca	Tuscany	1994	SE	28	Pisa

- 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:
  - $X \rightarrow A1 \ldots An$ iff $X \rightarrow A1 \ \ldots \ X \rightarrow An$
- What about the lhs:
  - Does $A1 \ldots An \rightarrow X$ imply $A1 \rightarrow X \ldots An \rightarrow X$?
  - Does $A1 \rightarrow X$ imply $A1..An \rightarrow X$?
- What does $\emptyset \rightarrow X$ mean?

Functional dependencies and keys

- Canonical dependencies:
  - $X \rightarrow A$ but not $X' \rightarrow A$, for any $X' \subset X$
- 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 $|\!-$ $X \rightarrow Y$), 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 |= F and F |- $X \rightarrow Y$, then r |= $X \rightarrow Y$

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:** $F \vdash X \rightarrow Y \Leftrightarrow Y \subseteq X_F^+$

Example

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

- StudCode^+ ={StudCode, Name, City, BirthYear, Region, Univ}
- $(\text{StudCode, Name})^+$ = {
- $(\text{Name,City})^+$ = {Name, City, Region, Univ}
- $(\text{StudCode, Subject})^+$
- $\emptyset^+$

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 $\varnothing \rightarrow A$ 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 $T1 \cup \ldots \cup Tn$ (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(<u>PkProduct</u>, ProductName, UnitPrice)
Invoices(<u>PkInvoiceNo</u>, Customer, Date, TotalPrice)
InvoiceLines(<u>FkInvoiceNo</u>, <u>LineNo</u>, 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

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

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

Group by elimination

Products(<u>PkProduct</u>, ProductName, UnitPrice)
Invoices(<u>PkInvoiceNo</u>, Customer, Date, TotalPrice)
InvoiceLines(<u>FkInvoiceNo</u>, <u>LineNo</u>, 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

nested correlated

select *
from studenti s
where exists (select * from exams e **where** e.sid=s.sid)

nested not correlated

select *
from students s
where s.id **in** (**select** e.sid **from** exams e)

unnested

select distinct s.*
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

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

SELECT *
FROM R
NATURAL JOIN
S;

A	B	C
1	a	x
3	c	y

Also called: natural **inner** join

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

SELECT *
FROM R
NATURAL LEFT JOIN
S;

A	B	C
1	a	x
2	b	
3	c	y

Also called: natural left **outer** join

Outer join: right, full

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

SELECT *
FROM R
NATURAL RIGHT JOIN
S;

A	B	C
1	a	x
3	c	y
5		z

Also called: natural right **outer** join

R

A	B
1	a
2	b
3	c

S

A	C
1	x
3	y
5	z

SELECT *
FROM R
NATURAL FULL JOIN
S;

A	B	C
1	a	x
2	b	
3	c	y
5		z

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);

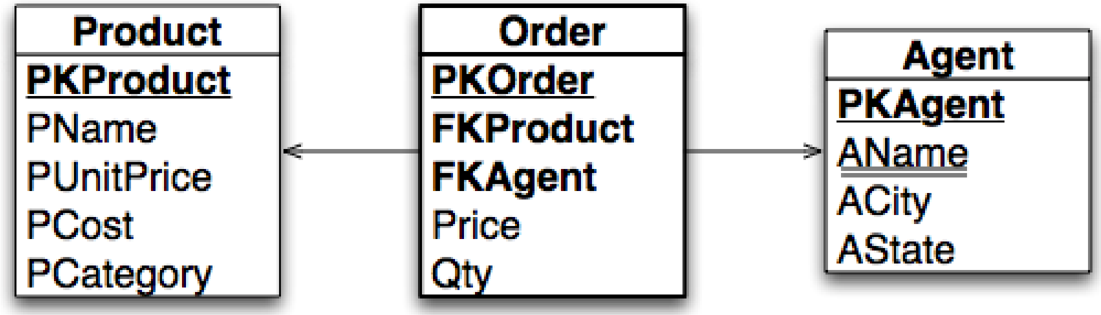
WHERE unnesting

- **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



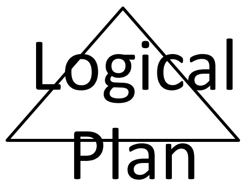
- CREATE VIEW TestView AS
  SELECT Price, AName
  FROM Order, Agent
  WHERE FKAgent = PKAgent;
- SELECT Price, AName
  FROM TestView
  WHERE Price = 1000;
- Can the query be transformed to avoid the use of the view?

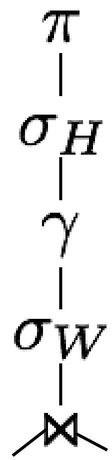
Temporary view

- 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

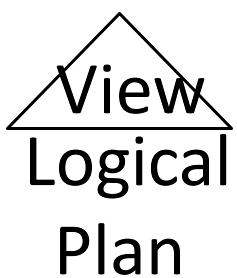
The approach:

Let a 

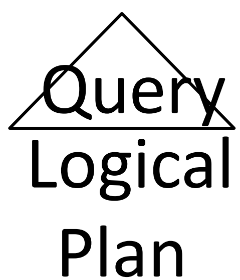
be 

View merging:

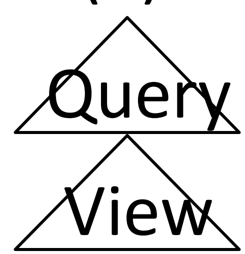
(1)



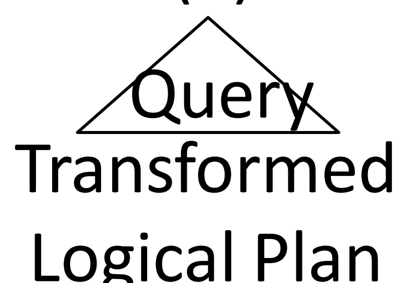
(2)

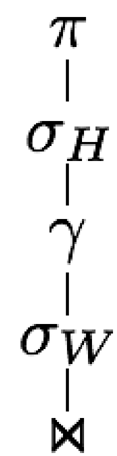


(3)



(4)





SQL without View

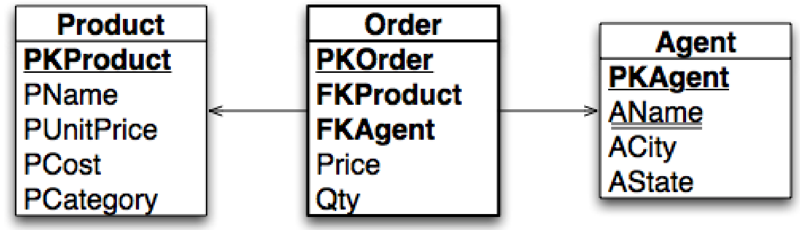
View merging: an equivalence rule

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

$$({}_{X_R}\gamma_F(R)) \underset{f_k=p_k}{\bowtie} S \equiv {}_{X_R \cup \mathcal{A}(S)}\gamma_F(R \underset{f_k=p_k}{\bowtie} S)$$

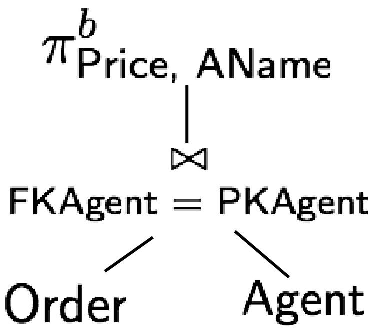
```
CREATE VIEW TestView AS
SELECT    Price, AName
FROM      Order, Agent
WHERE     FKAgent = PKAgent;
```

```
SELECT    Price, AName
FROM      TestView
WHERE     Price = 1000;
```

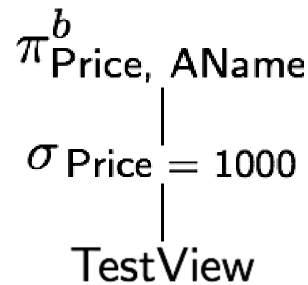


TestView =

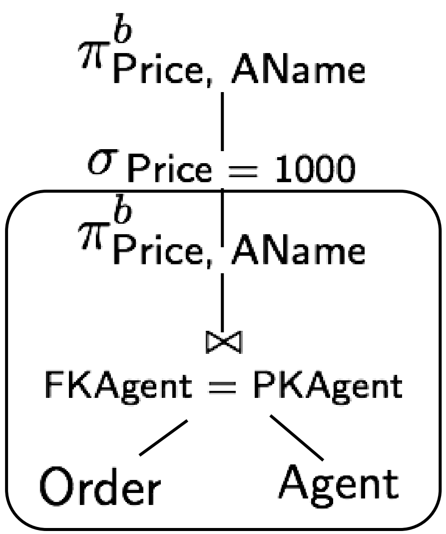
(1)



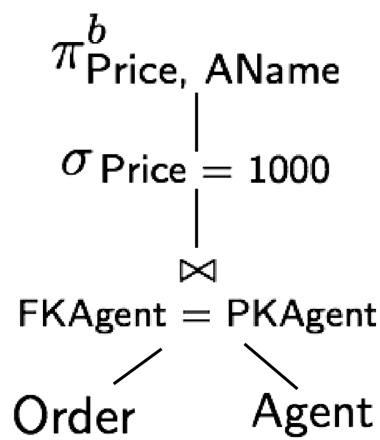
(2)



(3)

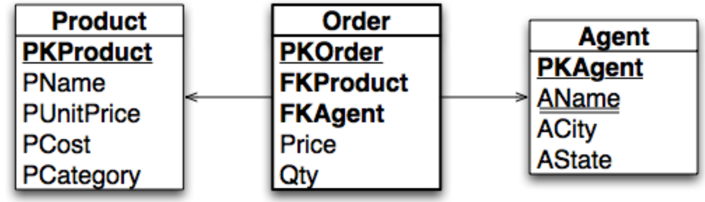


(4)



```
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';
```

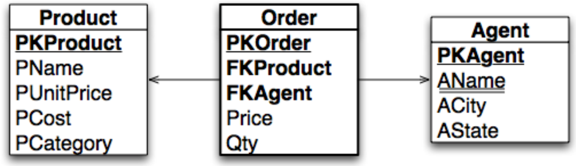


(1) $$\text{FKAgent\_GBY} = {}_{\text{FKAgent}}\gamma_{\text{COUNT(*) AS No}}(\text{Order})$$

(2) $$\pi^{b}_{\text{AName, No}}\left(\sigma_{\text{ACity} = \text{'Pisa'}}\left(\text{FKAgent\_GBY} \underset{\text{FKAgent} = \text{PKAgent}}{\bowtie} \text{Agent}\right)\right)$$

(3) $$\pi^{b}_{\text{AName, No}}\left(\sigma_{\text{ACity} = \text{'Pisa'}}\left(\boxed{{}_{\text{FKAgent}}\gamma_{\text{COUNT(*) AS No}}(\text{Order})} \underset{\text{FKAgent} = \text{PKAgent}}{\bowtie} \text{Agent}\right)\right)$$

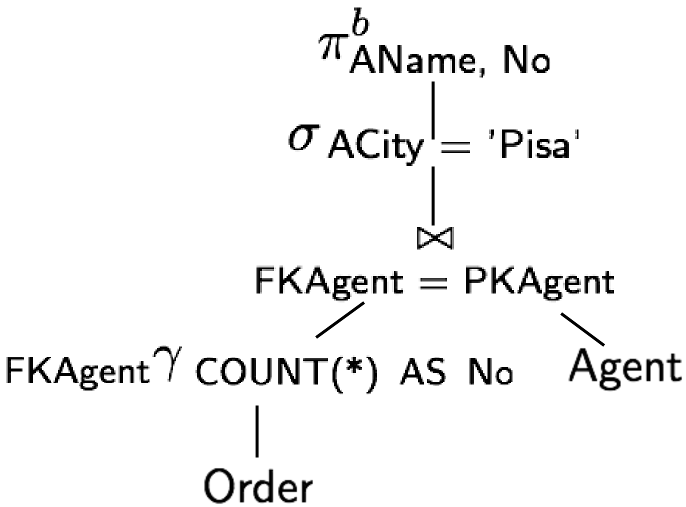
(4) $$({}_{X_R}\gamma_F(R)) \underset{f_k = p_k}{\bowtie} S \equiv {}_{X_R \cup \mathcal{A}(S)}\gamma_F(R \underset{f_k = p_k}{\bowtie} S)$$



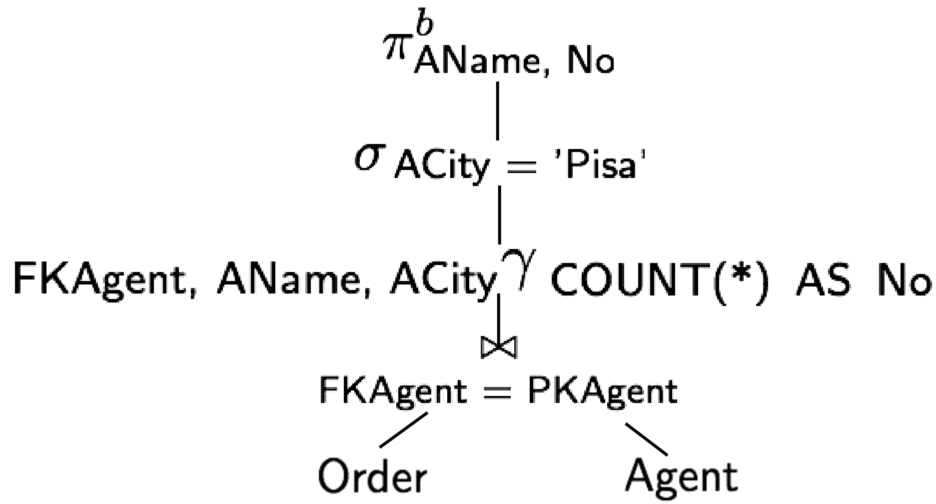

```
CREATE VIEW FKAgent_GBY AS
SELECT    FKAgent, COUNT(*) AS No
FROM      Order
GROUP BY  FKAgent;
```

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

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



(4)



$$({}_{X_R}\gamma_F(R)) \underset{f_k=p_k}{\bowtie} S \equiv {}_{X_R \cup \mathcal{A}(S)}\gamma_F(R \underset{f_k=p_k}{\bowtie} S)$$

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_{reg} and N_{pag} for each relation.
  - N_{key} and N_{leaf} for each index.
  - min/max values for each index key.
  - ... Histograms
- The Catalog is updated with the command **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), a 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!

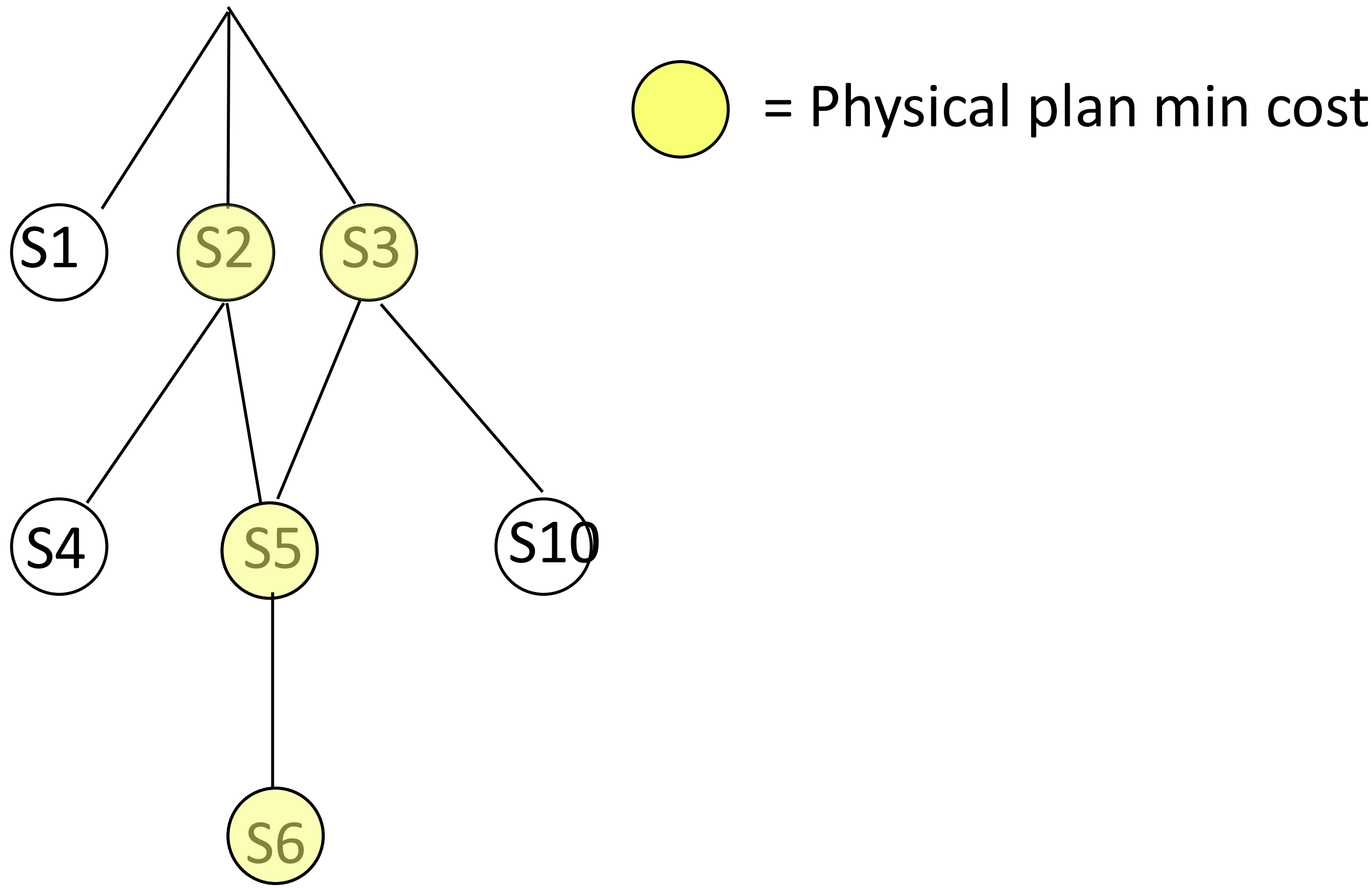
Full search

R Join S Join T

One relation

Two relations

Three relations



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

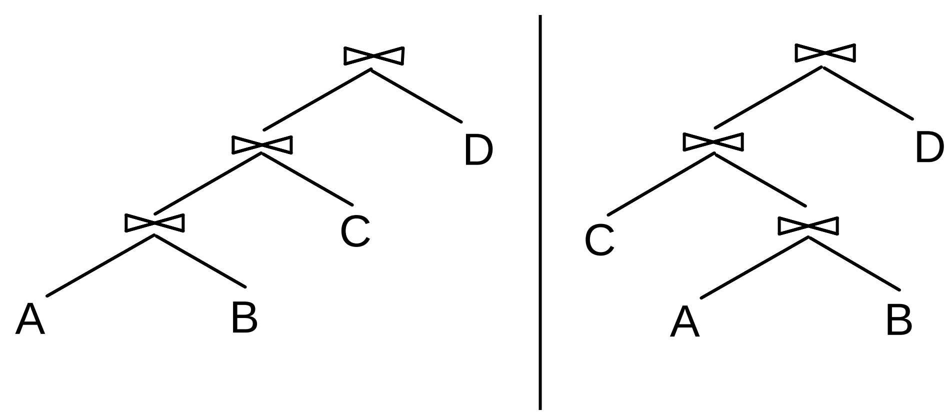
        for each *P* join *P*’, put the best tree in *Plans*

        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($\underline{N}$, D, T, $\underline{C}$), with indexes on C and T

S($\underline{C}$, 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’;

$$\pi^{b}_{S.C,\ S.O}(\sigma_{E = 13 \wedge T = 'AA'}(S \bowtie R))$$

$$\pi^{b}_{S.C,\ S.O}(\boxed{\sigma_{E = 13}(S)} \bowtie \boxed{\sigma_{T = 'AA'}(R)})$$

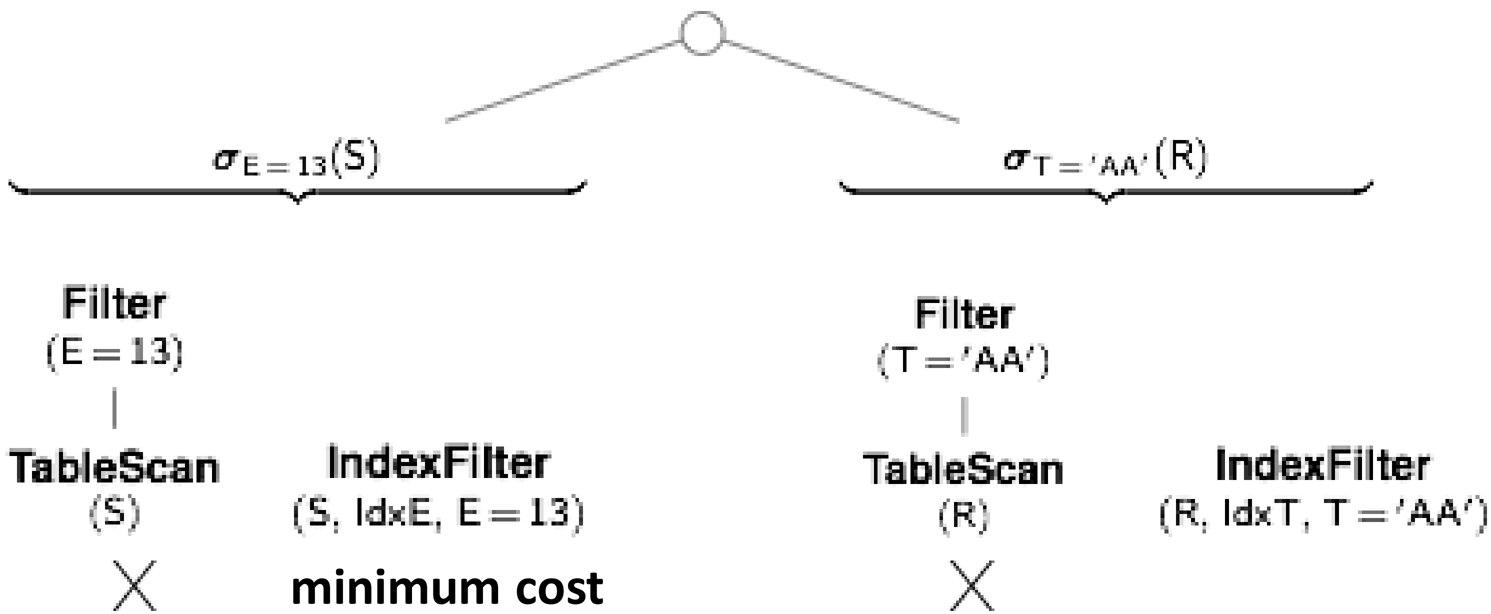
Example

R($\underline{N}$, D, T, $\underline{C}$), with indexes on C and T

S($\underline{C}$, O, E), with indexes on C and E

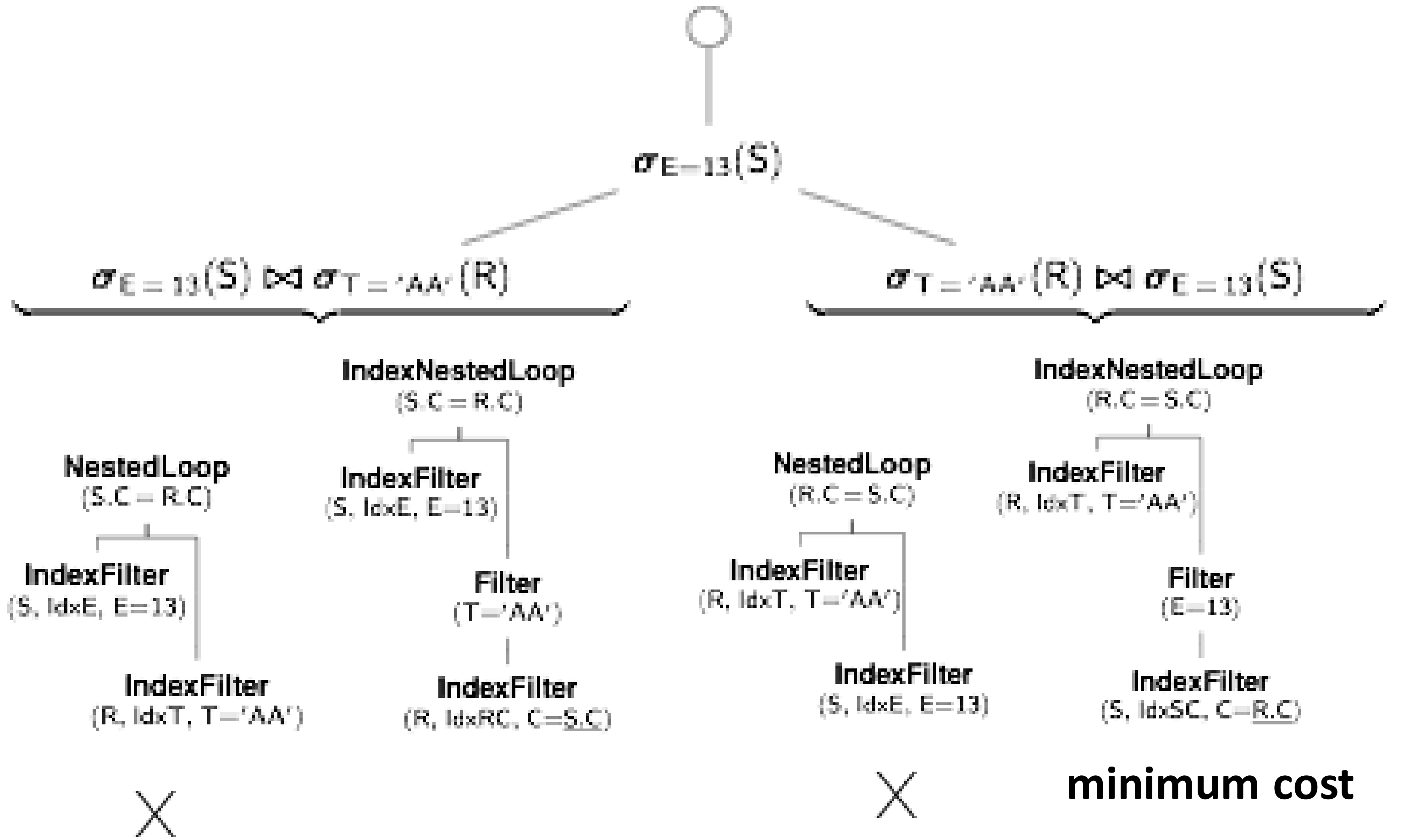
$$\pi^{b}_{S.C,\ S.O}\left(\boxed{\sigma_{E=13}(S)} \bowtie \boxed{\sigma_{T='AA'}(R)}\right)$$

Physical plans for subexpression on relations



Example

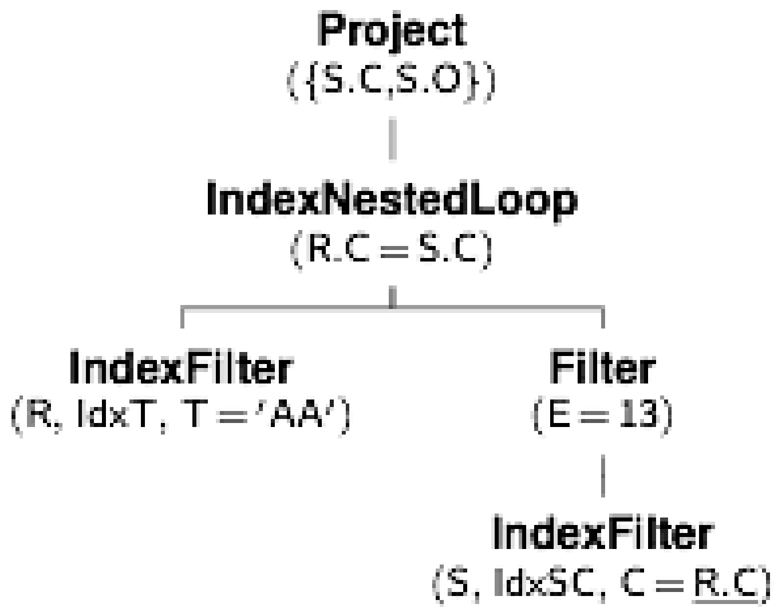
$$\pi^{b}_{S.C,\ S.O}\left(\ \boxed{\sigma_{E\,=\,13}(S)\ \bowtie\ \sigma_{T\,=\,'AA'}(R)}\ \right)$$



Example

$$\pi^{b}_{S.C,\ S.O}\left(\sigma_{E\,=\,13}(S) \bowtie \sigma_{T\,=\,'AA'}(R) \right)$$

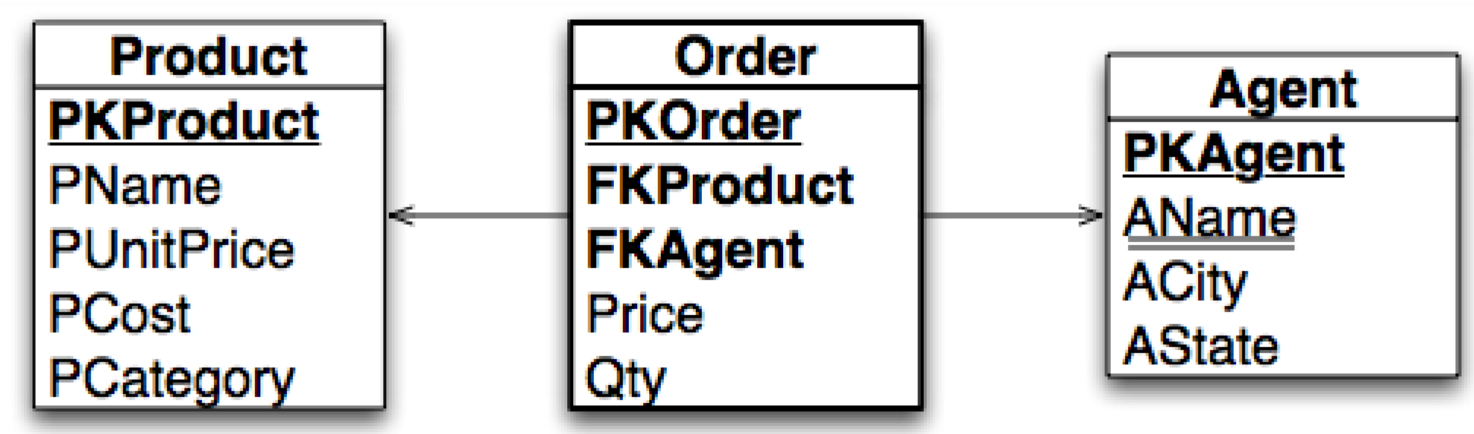
Final physical plan



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

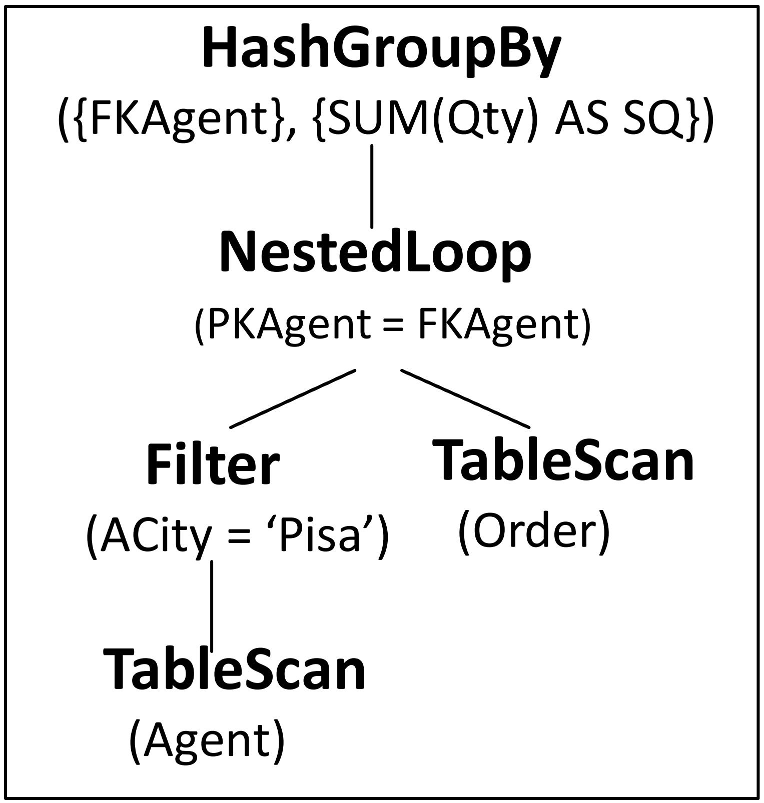
Example



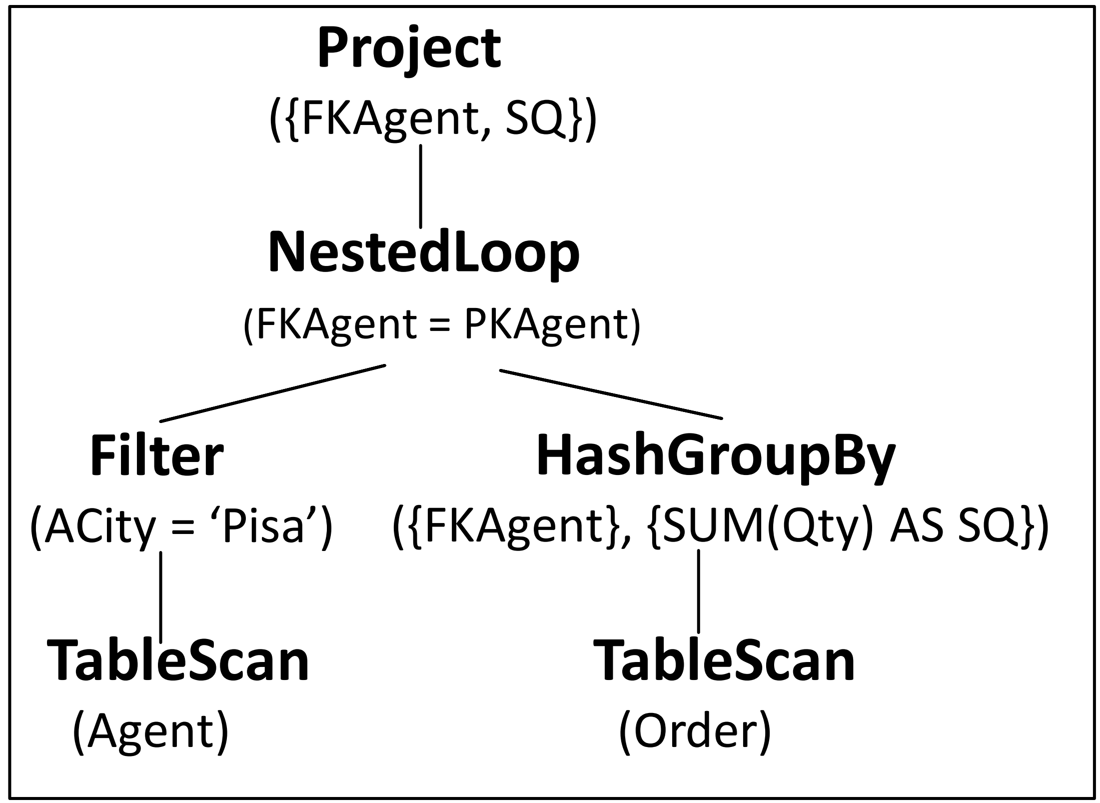

```
SELECT      FKAgent, SUM(Qty) AS SQ
FROM        Order, Agent
WHERE       FKAgent = PKAgent AND ACity = 'Pisa'
GROUP BY    FKAgent;
```

Pre-grouping

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



Standard Physical Plan



Physical Plan with the Pre-Grouping

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

$$_{X}\gamma_{F}(R \underset{f_k = p_k}{\bowtie} S)$$

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

$$_{X}\gamma_{F}(R \underset{f_k = p_k}{\bowtie} S) \stackrel{?}{\equiv} \ldots((_{X'}\gamma_{F'}(R)) \underset{f_k = p_k}{\bowtie} S)$$

Grouping equivalence rules: σ

$$\sigma_\phi({}_X\gamma_F(E)) \stackrel{?}{\equiv} {}_X\gamma_F(\sigma_\phi(E))$$

Two cases to consider for the selection

1) $\sigma_{\phi_X}({}_X\gamma_F(E)) \equiv {}_X\gamma_F(\sigma_{\phi_X}(E))$ In SQL ?

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

$\mathsf{AGG} = \mathsf{COUNT, SUM, MIN, MAX, AVG}$

Bad news: two cases only

$$\sigma_{\mathsf{Mb} \geq \mathsf{v}}({}_X\gamma_{\mathsf{MAX(b)\ AS\ Mb}}(E)) \equiv {}_X\gamma_{\mathsf{MAX(b)\ AS\ Mb}}(\sigma_{\mathsf{b} \geq \mathsf{v}}(E))$$

$$\sigma_{\mathsf{mb} \leq \mathsf{v}}({}_X\gamma_{\mathsf{MIN(b)\ AS\ mb}}(E)) \equiv {}_X\gamma_{\mathsf{MIN(b)\ AS\ mb}}(\sigma_{\mathsf{b} \leq \mathsf{v}}(E))$$

Grouping equivalence rules

Assume that X→Y:

$$ {}_{X}\gamma_{F}(E) \equiv \pi^{b}_{X\cup F}({}_{X\cup Y}\gamma_{F}(E)) $$

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

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

PKOrder	FKAgent	...	PKAgent	AName	ACity	...
1	1	...	1	Rossi	Pisa	...
2	2	...	2	Verdi	Firenze	...
3	1	...	1	Rossi	Pisa	...
4	2	...	2	Verdi	Firenze	...

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

$$_X\gamma_F(R \underset{f_k=p_k}{\bowtie} S)$$

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

$$_X\gamma_F(R \underset{f_k=p_k}{\bowtie} S) \stackrel{?}{\equiv} \ldots((_{X'}\gamma_{F'}(R)) \underset{f_k=p_k}{\bowtie} S)$$

Three cases

The invariant grouping rule

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

$$_X\gamma_F(R \bowtie_{C_j} S) \equiv \pi^b_{X \cup F}((_{X \cup \mathcal{A}(C_j) - \mathcal{A}(S)}\gamma_F(R)) \bowtie_{C_j} S)$$

if the following conditions are true:

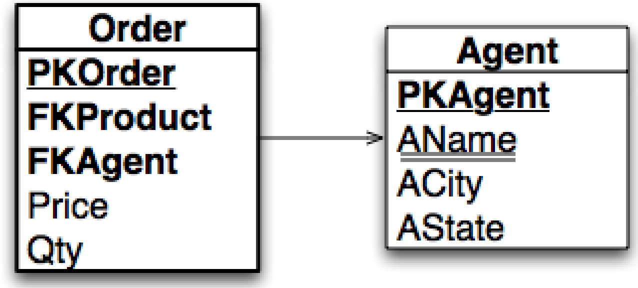
1. C_j |- $X \rightarrow A(S)$: in every group, only one line from S

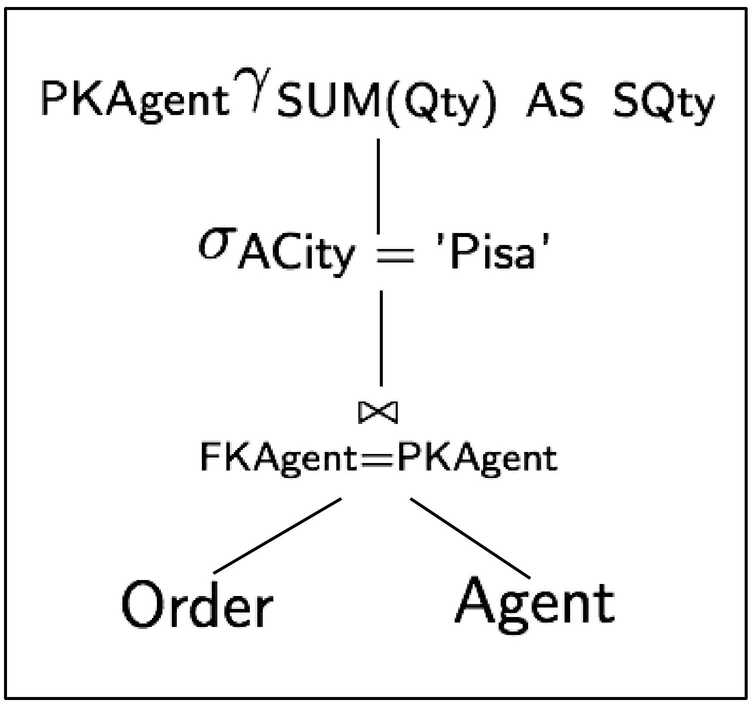
   in practice: C_j is $f_k = p_k$, with f_k in R, p_k key for S, $X \longrightarrow f_k$

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

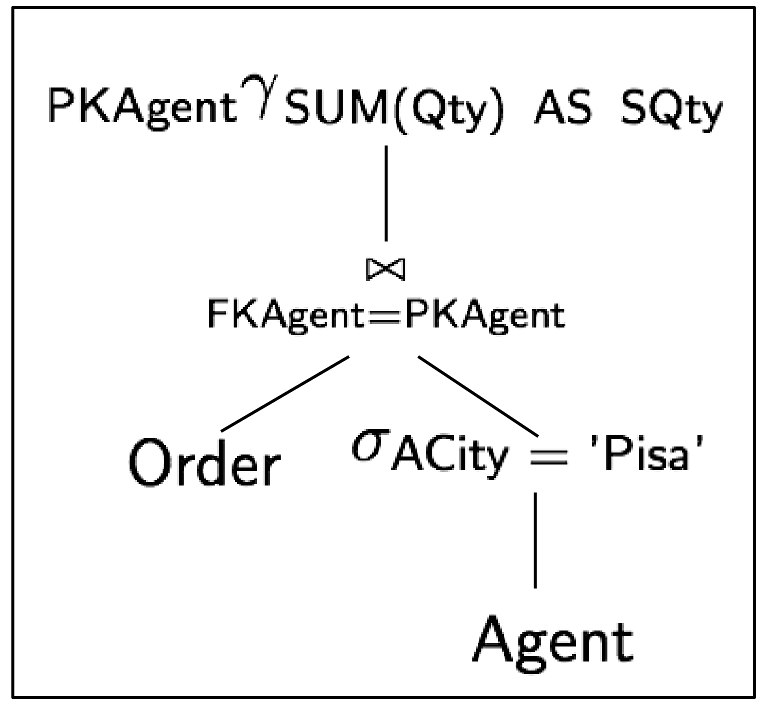
Example

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

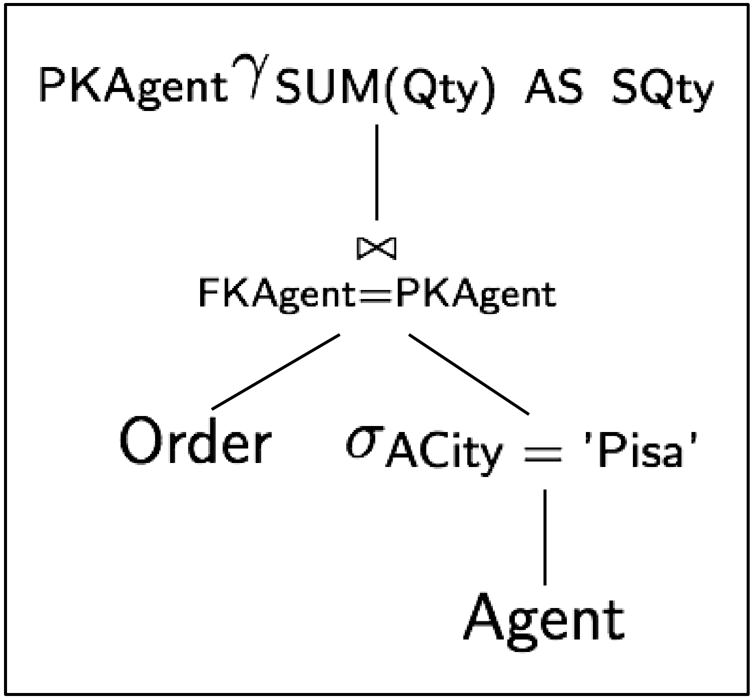




≡

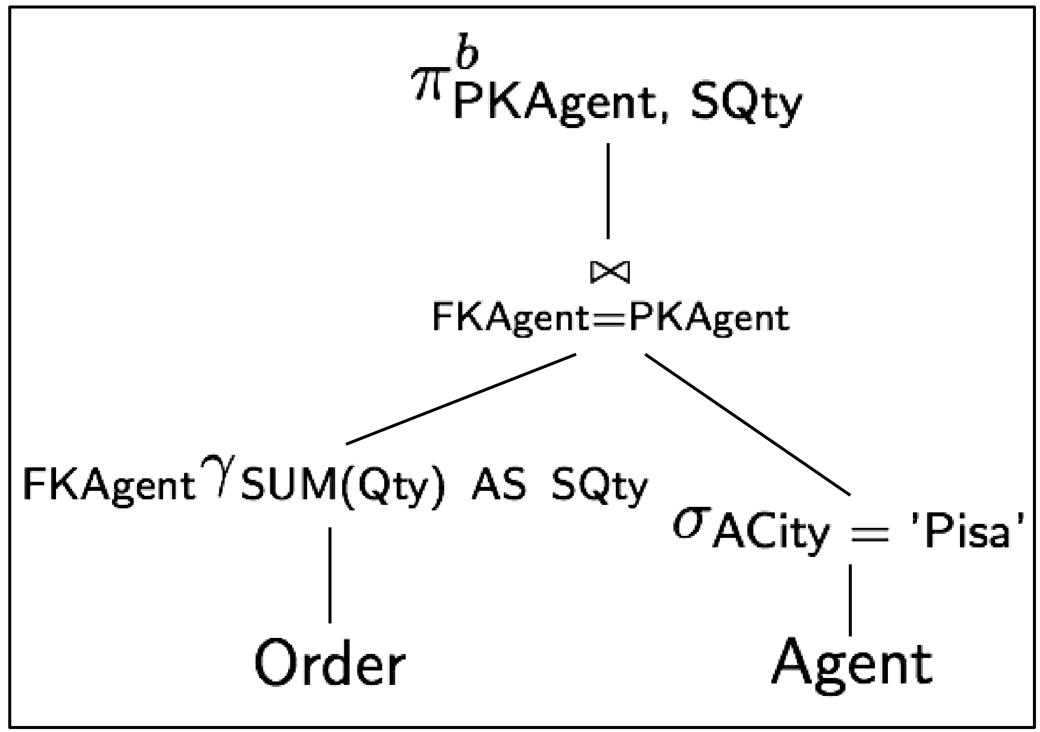


Example

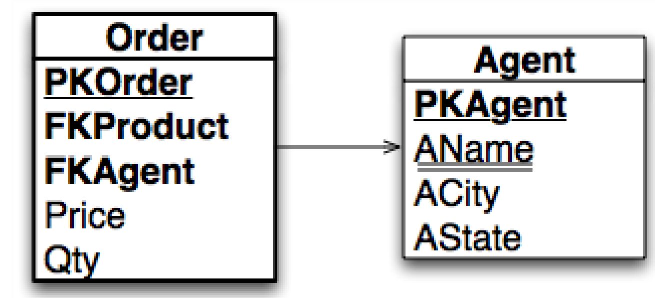


$$ {}_X\gamma_F(R \underset{C_j}{\bowtie} S) \equiv $$

$$ \pi^b_{X\cup F}(({}_{X\cup\mathcal{A}(C_j)-\mathcal{A}(S)}\gamma_F(R)) \underset{C_j}{\bowtie} S) $$



Tests




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

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

```sql
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