Cloud Data Management:
Relational Databases vs MapReduce

Lecture 5

Recap

- MapReduce introduced by Google
  - Simple programming model for building distributed applications that process vast amounts of data
  - Runtime for executing jobs on large clusters in a reliable, fault-tolerant manner

- Hadoop makes MapReduce broadly available
  - HDFS becomes central data repository
  - Becomes Defacto standard for batch processing

New applications, new workloads

- MapReduce originally designed for batch analytics
  - Latency-insensitive: jobs that run for hours
    - Sequential scans of Petabytes of data
  - Built for fault tolerance across thousands of commodity servers
    - Focus on faults during query rather than recovery after updates

- Hadoop starts being used for interactive computations, e.g., ad-hoc analytics
  - Hive and Pig frameworks with SQL-over-Hadoop drive this trend

- But SQL and analytics was the stronghold of relational database engines

“MapReduce: A major step backwards” – Dewitt, Stonebraker

Role of a database system

- Database: **integrated, shared** data collection
- Integrated
  - Eliminate needless redundancy
  - Maintain strong consistency
- Shared
  - Application written by <u>programmers</u> in multiple languages
  - <u>End-users</u> who use applications, forms, CLI to interact
- Database systems shield users from
  - How data is stored (bits & bytes, 1 vs N files, 1 vs N disks…)
  - How data is accessed (btree, hashtable, scan, …)

What is a data model?

- Collection of application-visible constructs
  - Describe data in application & storage agnostic way
- Constructs to describe structural aspects
  - How do applications perceive the data?
  - Ex: table, graph, associative array…
- Constructs to describe manipulation aspects
  - What operators can applications use?
  - Ex: join, traverse, lookup…
- Constructs to describe data integrity aspects
  - How do we ensure that data manipulation is “correct”?

Relational Model: Structural aspect

- Database = set of named **relations** (or **tables**)
- Each relation has a set of named **attributes** (or **columns**)
- Each **tuple** (or **row**) has a value for each attribute
- Each attribute has a **type** (or **domain**)
  - integer, real, string, file formats (jpeg,…), enumerated and many more

Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Colleges

name	location	strength
MIT	USA	10000
Oxford	UK	22000
EPFL	CH	9000
…	…	…

Relational Model: Structural aspect

- **Relation Schema**: relation name + field names + field domains
  - Students(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real)
- **Relation Instance**: contents at a given point in time
  - *set* of rows or *tuples*. (all rows are distinct with no specific ordering)
  - Cardinality: # rows, Arity or degree: # attributes
- **Database Schema**: collection of relation schemas
- **Database Instance**: collection of relation instances

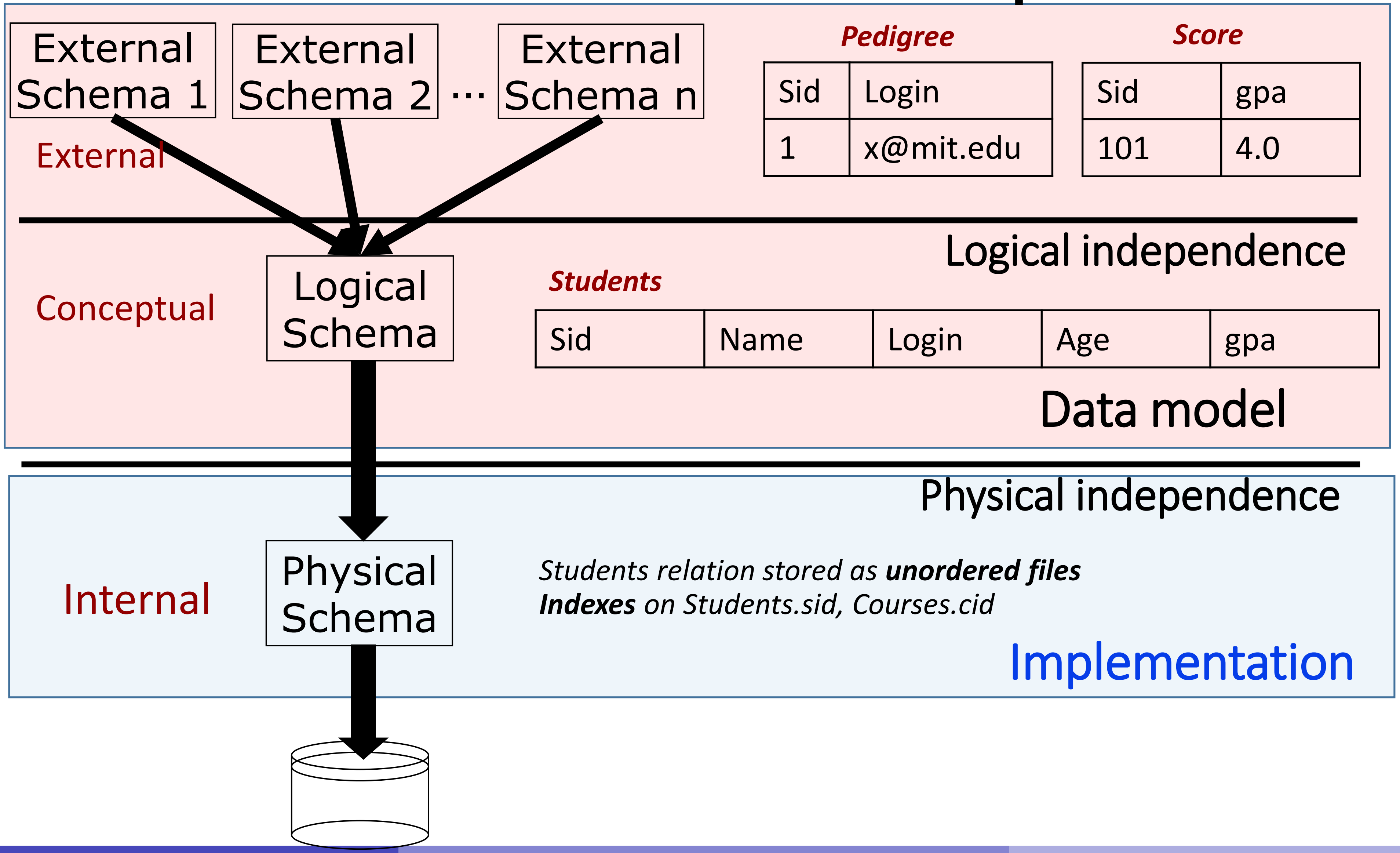
Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Colleges

name	location	strength
MIT	USA	10000
Oxford	UK	22000
EPFL	CH	9000
…	…	…

Relational model & data independence



Pedigree

Sid	Login
1	x@mit.edu

Score

Sid	gpa
101	4.0

Students

Sid	Name	Login	Age	gpa

*Students relation stored as **unordered files***
***Indexes** on Students.sid, Courses.cid*

Relational Model: Integrity Aspect

- Relational model provides **Integrity Constraints**
  - condition specified on schema that restricts the data that can be stored in ***any*** instance
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A ***legal*** instance of a relation is one that satisfies all specified ICs
  - DBMS should not allow illegal instances.
- With ICs, stored data is more faithful to real-world meaning
  - Avoids data entry errors, too!

Relational Model: **Keys**

- Attribute whose value is unique in each tuple
- Or set of attributes whose combined values are unique
- Keys specify **key constraint**
  - Enforced when tuples are inserted/updated

Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Colleges

name	location	strength
MIT	USA	10000
Oxford	UK	22000
EPFL	CH	9000
…	…	…

Relational Model: **Foreign Keys**

- Set of fields in one relation that `refer' to a tuple in another relation (like a pointer)
- Foreign keys specify **Foreign Key Constraint**
  - FK must correspond to the primary key of the other relation
- If all foreign key constraints are enforced, **referential integrity** is achieved (i.e., no dangling references.)

Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Enrolled

cid	sid	grade
Carnatic101	53666	C
Raggae203	50000	B
Topology112	53666	A
…	…	…

Relational Model: Manipulation Aspect

- **Query languages**: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:
  - ***Relational Algebra***: More operational, very useful for representing execution plans.
  - **Relational Calculus:** Lets users describe what they want, rather than how to compute it. (Non-procedural, *declarative*.)

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run over any legal instance)
  - The schema for the *result* of a given query is also fixed. It is determined by the definitions of the query language constructs.

Example Schema and Instances

- `Boats(`*bid*`: integer, `*bname*`: string, `*color*``: string)`
- `Sailors(`*sid*`: integer, `*sname*`: string, `*rating*`: integer, `*age*`: real)`
- `Reserves(`*sid*`: integer, `*bid*`: integer, `*day*`:date)`

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Relational Algebra: 5 Basic Operations

- **Selection** (σ) Selects a subset of *rows* from relation (horizontal).
- **Projection** (π) Retains only wanted *columns* from relation (vertical).
- **Cross-product** ($\times$) Allows us to combine two relations.
- **Set-difference** ($-$) Tuples in r1, but not in r2.
- **Union** ($\cup$) Tuples in r1 and/or in r2.

Since each operation returns a relation, operations can be *composed!* (Algebra is “closed”).

Selection Operator: (σ)

- Selects rows that satisfy *selection condition*.
- ***Output schema*** of result is same as that of the input relation

S2

sid	sname	rating	age
~~28~~	~~yuppy~~	~~9~~	~~35.0~~
31	Lubber	8	55.5
44	guppy	5	35.0
~~58~~	~~Rusty~~	~~10~~	~~35.0~~

$\sigma_{rating<9}(S2)$

Output

sid	sname	rating	age
31	Lubber	8	55.5
44	guppy	5	35.0

S2

sid	sname	rating	age
~~28~~	~~yuppy~~	~~9~~	~~35.0~~
31	Lubber	8	55.5
~~44~~	~~guppy~~	~~5~~	~~35.0~~
~~58~~	~~Rusty~~	~~10~~	~~35.0~~

$\sigma_{\substack{rating<9\wedge \\ age>50}}(S2)$

Output

sid	sname	rating	age
31	Lubber	8	55.5

Projection Operator (π)

- Retains only attributes that are in the *projection list.*
- **Output schema** is exactly the fields in the projection list, with the same names that they had in the input relation.

S2

~~sid~~	sname	rating	~~age~~
~~28~~	yuppy	9	~~35.0~~
~~31~~	Lubber	8	~~55.5~~
~~44~~	guppy	5	~~35.0~~
~~58~~	Rusty	10	~~35.0~~

$$\pi_{sname,rating}(S2)$$

Output

sname	rating
yuppy	9
Lubber	8
guppy	5
Rusty	10

Projection Operator (π): Duplicate Elimination

- Relational algebra is set based while SQL is bag (multiset) based
- Projection operator *eliminates duplicates*

$S2$

~~sid~~	~~sname~~	~~rating~~	age
~~28~~	~~yuppy~~	~~9~~	35.0
~~31~~	~~Lubber~~	~~8~~	55.5
~~44~~	~~guppy~~	~~5~~	35.0
~~58~~	~~Rusty~~	~~10~~	35.0

$$\pi_{age}(S2)$$

$Output$

age
35.0
55.5

Composing multiple operators

- Output of one operator can become input to another operator

$S2$

~~sid~~	sname	rating	~~age~~
~~23~~	yuppy	9	~~35.0~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~
~~44~~	~~guppy~~	~~5~~	~~35.0~~
~~58~~	Rusty	10	~~35.0~~

$Output$

sname	rating
yuppy	9
Rusty	10

$$\pi_{sname,rating}\left(\sigma_{rating>8}(S2)\right)$$

Union and Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - “Corresponding” fields have the same type.

Union operator (U)

$S1$

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$S2$

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$S1 \cup S2$

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

Set Difference Operator (–)

$S1$

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$S2$

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$S1 - S2$

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0

$S2 - S1$

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
44	guppy	5	35.0

Cross-Product (×)

- S1 x R1: Each row of S1 paired with each row of R1.

- **Result schema** has one field per field of S1 and R1, with field names “inherited” if possible.
  - *May have a naming conflict*: Both S1 and R1 have a field with the same name.
  - In this case, can use the *renaming operator*:

$$\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$$

Call C the result of S1×R1 and respectively rename the 1st & 5th fields of C to sid1 & sid2

Cross-Product Example

$\boldsymbol{S1}$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$\boldsymbol{R1}$

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$\boldsymbol{S1 \times R1}$

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

$\rho_{1 \to sid1, 5 \to sid2}(\boldsymbol{S1 \times R1})$

sid1	sname	rating	age	sid2	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Compound Operator: Join

- Joins are compound operators involving cross product, selection, and (sometimes) projection.
- Most common type of join is a **natural join** (often just called "join"). $R \bowtie S$ conceptually is:
  - Compute $R \times S$
  - Select rows where attributes that appear in both relations have equal values
  - Project all unique attributes and one copy of each of the common ones.
- Note: Usually done much more efficiently than this.
- Useful for putting "normalized" relations back together.

Natural Join Example

$\boldsymbol{S1}$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$\boldsymbol{R1}$

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$\pi_{S1.sid,sname,\ldots}\left(\sigma_{S1.sid=R1.sid}\left(\boldsymbol{S1} \times \boldsymbol{R1}\right)\right)$

sid	sname	rating	age	~~sid~~	bid	day
22	Dustin	7	45.0	~~22~~	101	10/10/96
~~22~~	~~Dustin~~	~~7~~	~~45.0~~	~~58~~	~~103~~	~~11/12/96~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~	~~22~~	~~101~~	~~10/10/96~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~	~~58~~	~~103~~	~~11/12/96~~
~~58~~	~~Rusty~~	~~10~~	~~35.0~~	~~22~~	~~101~~	~~10/10/96~~
58	Rusty	10	35.0	~~58~~	103	11/12/96

$\boldsymbol{S1} \bowtie \boldsymbol{R1}$

sid	sname	rating	age	bid	day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Condition Join or Theta-Join

$$R \bowtie_C C = \sigma_C(R \times S)$$

- **Output schema** same as that of cross-product.
- May have fewer tuples than cross-product.

$S1$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$R1$

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{S1.sid<R1.sid} R1$$

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Equi-Join

- Special case of theta-join: condition *c* contains only conjunction of *equalities*.
- Find **all pairs** of sailors in S2 who have same age.

S2

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

- $\boldsymbol{S1} \bowtie_{S1.age=S2.age} \boldsymbol{S2}$
- $\sigma_{\boldsymbol{sid1!=sid2}} \left(\boldsymbol{S1} \bowtie_{S1.age=S2.age} (\boldsymbol{S2}) \right)$

Grouping and Aggregation

- Grouping and Aggregation: γX (R)
  - Given a relation R, partition its tuples according to their values in one set of attributes G
    - The set G is called the grouping attributes
  - Then, for each group, aggregate the values in certain other attributes
    - Aggregation functions: SUM, COUNT, AVG, MIN, MAX, ...
- In the notation, X is a list of elements that can be:
  - A grouping attribute
  - An expression θ(A), where θ is one of the (five) aggregation functions and A is an attribute NOT among the grouping attributes

Grouping and Aggregation: Example

- Let’s work with an example
  - Imagine that a social-networking site has a relation `Friends(User, Friend)`
  - The tuples are pairs *(a, b)* such that b is a friend of a
  - *Query: compute the number of friends each member has*

- $\gamma_{\mathit{User,\ COUNT(Friend)}}$ *(Friends)*
  - This operation groups all the tuples by the value in their first component
  - There is one group for each user
  - Then, for each group, it counts the number of friends

Renaming Operator (ρ)

- Renames the list of attributes specified in the form of oldname → newname or position → newname
- **Output schema** is same as input except for the renamed attributes.
- Returns same tuples as input
- Can also be used to rename the name of the output relation

Boats

<u>bid</u>	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

$\rho_{bname \rightarrow boatname, color \rightarrow boatcolor}(Boats)$

<u>bid</u>	boatname	boatcolor
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

$\rho_{2 \rightarrow boatname, 3 \rightarrow boatcolor}(Boats)$

Relational Algebra: Summary

Formal foundation for real query languages

- Helps represent and reason about execution plans

5 basic operators forming a closed algebra

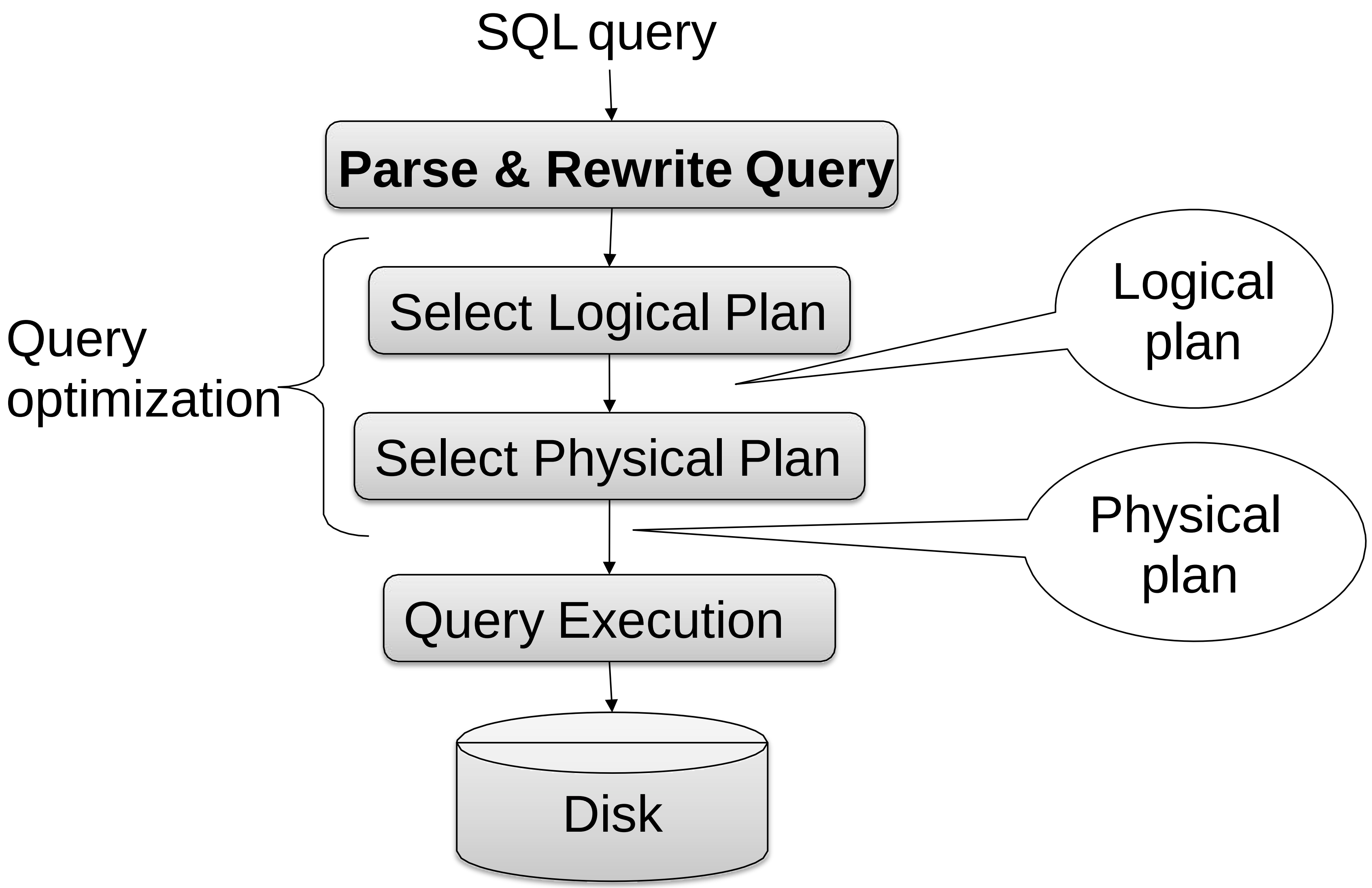
- Selection, projection, cross-product, union, set difference

Compound operators

- Useful shorthands like join and division
- Can be expressed with basic operators
- But enable faster query execution

Query Processing

Steps in Query Processing



Query parsing & transformation

A Query:

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

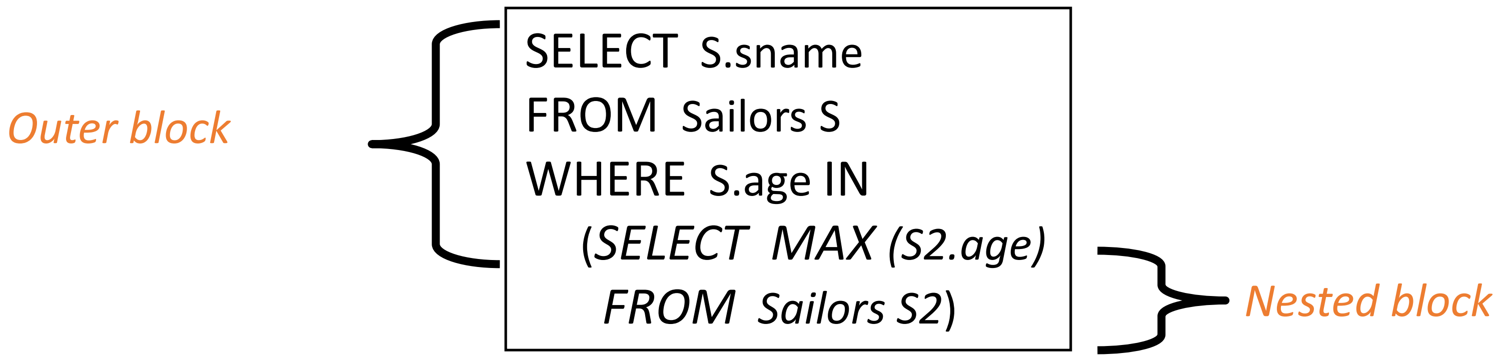
1. Query first broken into “blocks”
2. Each block converted to relational algebra

Step 1: Break query into Query Blocks

- Query block = unit of optimization

- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple

  (This is an over-simplification, but serves for now)



```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2)
```

Step 2: Converting query block into relational algebra expression

```
SELECT  S.sid
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
```

$$\pi_{\text{S.sid}}(\sigma_{\text{B.color} = \text{"red"}}(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}))$$

Relational Algebra Equivalences

- *Selections*: $\sigma_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}\left(\ldots\left(\sigma_{c_n}(R)\right)\right)$ (*Cascade*)

  $\sigma_{c_1}\left(\sigma_{c_2}(R)\right) \equiv \sigma_{c_2}\left(\sigma_{c_1}(R)\right)$ (*Commute*)

- *Projections:* $\pi_{a_1}(R) \equiv \pi_{a_1}\left(\ldots\left(\pi_{a_n}(R)\right)\right)$ *(Cascade)*

  a_i is a set of attributes of R and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$

- These equivalences allow us to ‘push’ selections and projections ahead of joins.

Another Equivalence

- A projection commutes with a selection that only uses attributes retained by the projection

$$\pi_{\text{age, rating, sid}} (\sigma_{\text{age}<18 \wedge \text{rating}>5} (\text{Sailors}))$$

$$\leftrightarrow \sigma_{\text{age}<18 \wedge \text{rating}>5} (\pi_{\text{age, rating, sid}} (\text{Sailors}))$$

Equivalences Involving Joins

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \qquad \textit{(Associative)}$$

$$(R \bowtie S) \equiv (S \bowtie R) \qquad \textit{(Commutative)}$$

- These equivalences allow us to choose different join orders

Examples …

$$\sigma_{age<18 \wedge rating>5} (Sailors)$$

$$\leftrightarrow \sigma_{age<18} (\sigma_{rating>5} (Sailors))$$

$$\leftrightarrow \sigma_{rating>5} (\sigma_{age<18} (Sailors))$$

~~$\pi_{age,rating} (Sailors) \leftrightarrow \pi_{age} (\pi_{rating} (Sailors))$~~ (??)

$$\pi_{age,rating} (Sailors) \leftrightarrow \pi_{age,rating} (\pi_{age,rating,sid} (Sailors))$$

Mixing Joins with Selections & Projections

- Converting selection + cross-product to join

$$\sigma_{S.sid = R.sid} \text{(Sailors x Reserves)}$$

$$\leftrightarrow \text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves}$$

- Selection on just attributes of S commutes with R $\bowtie$ S

$$\sigma_{S.age<18} (\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

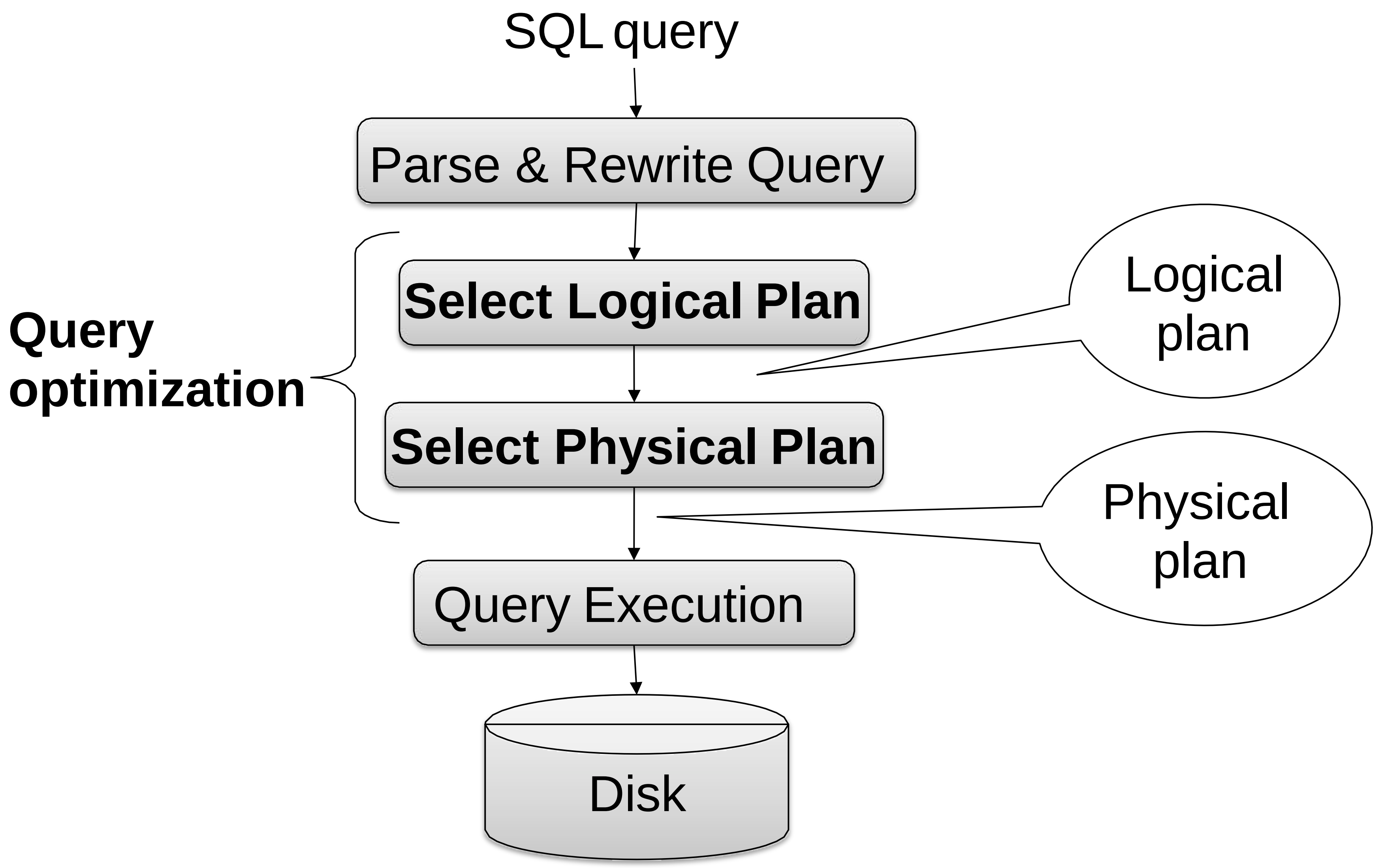
$$\leftrightarrow (\sigma_{S.age<18} (\text{Sailors})) \bowtie_{S.sid = R.sid} \text{Reserves}$$

- We can also "push down" projection (*but be careful…*)

$$\pi_{S.sname} (\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

$$\leftrightarrow \pi_{S.sname} (\pi_{sname,sid}(\text{Sailors}) \bowtie_{S.sid = R.sid} \pi_{sid}(\text{Reserves}))$$

Steps in Query Processing



We know…

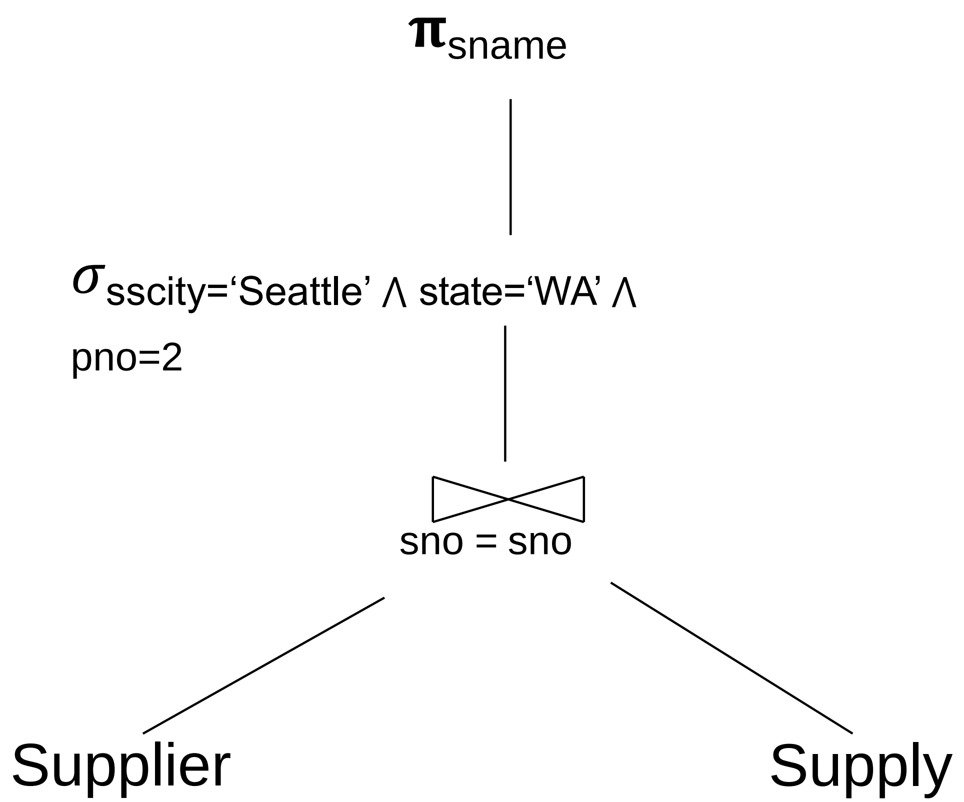
```
Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
```

For each SQL query….

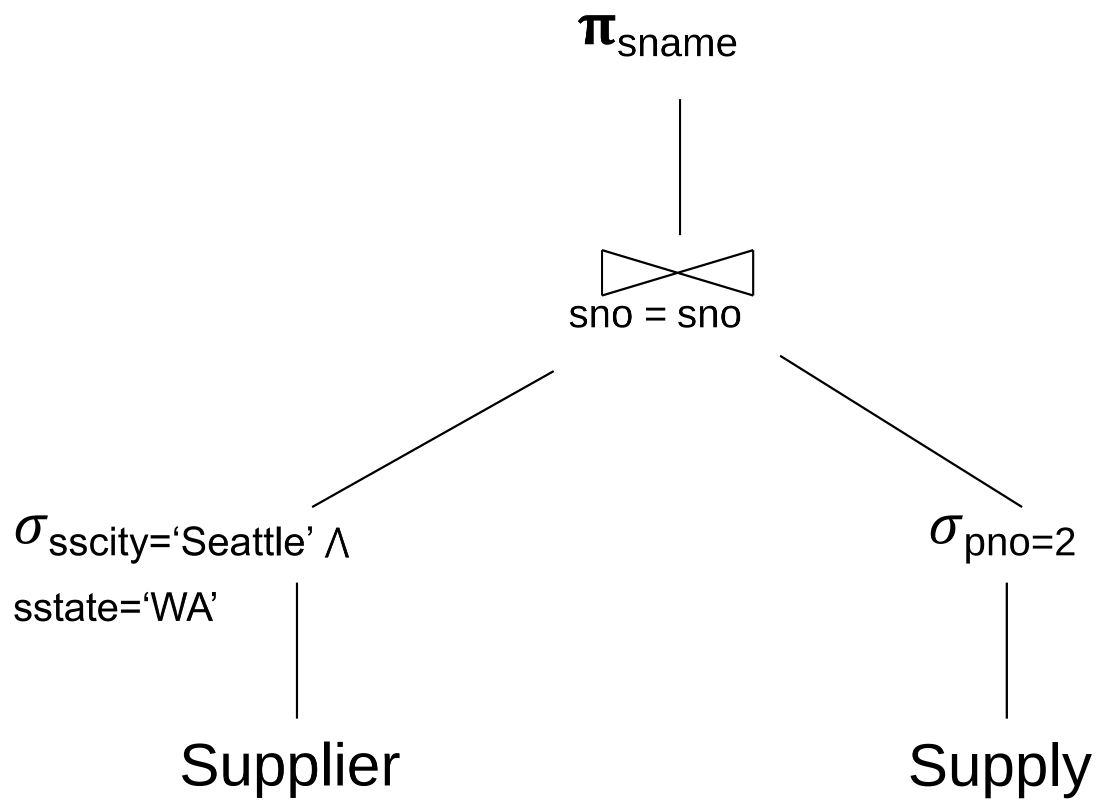
```
SELECT S.sname
FROM Supplier S, Supply U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND  S.sno  =  U.sno
AND  U.pno  =  2
```

There exist many logical query plans…

Example Query: Logical Plan 1


π_{sname}
$\sigma_{sscity=\text{'Seattle'} \land state=\text{'WA'} \land pno=2}$
$\bowtie$
sno = sno
Supplier
Supply

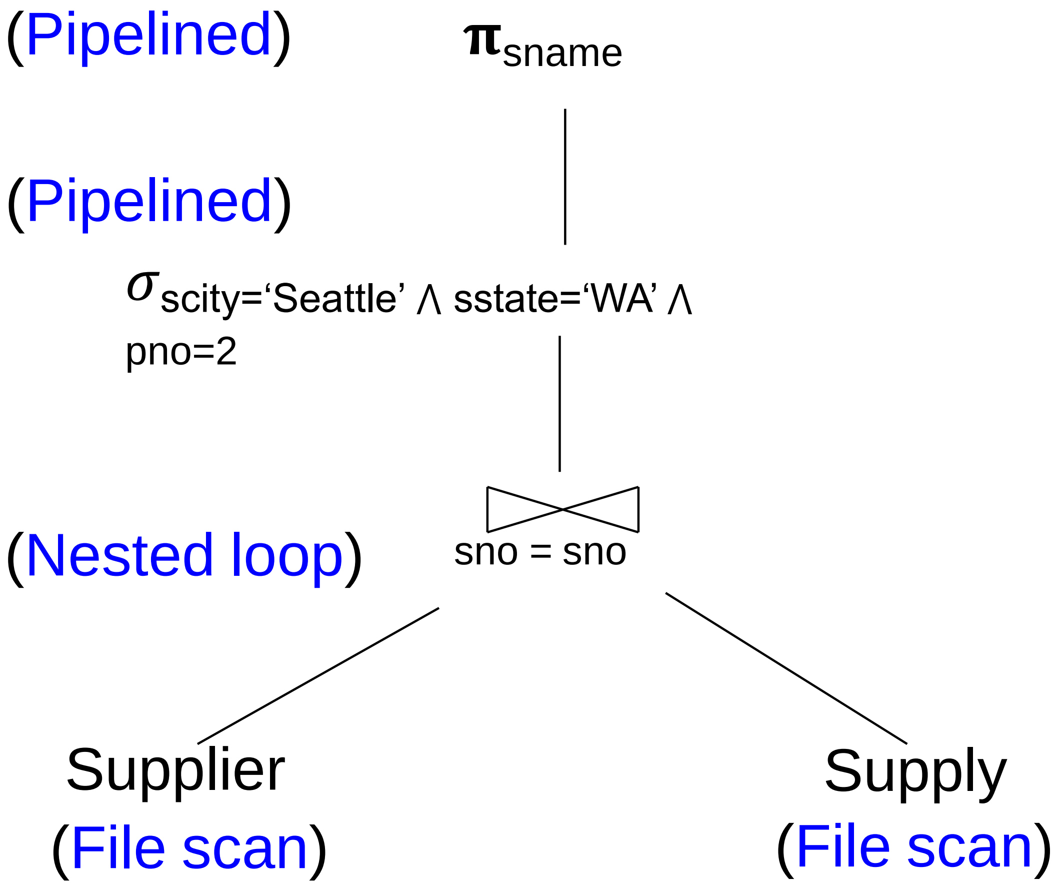
Example Query: Logical Plan 2



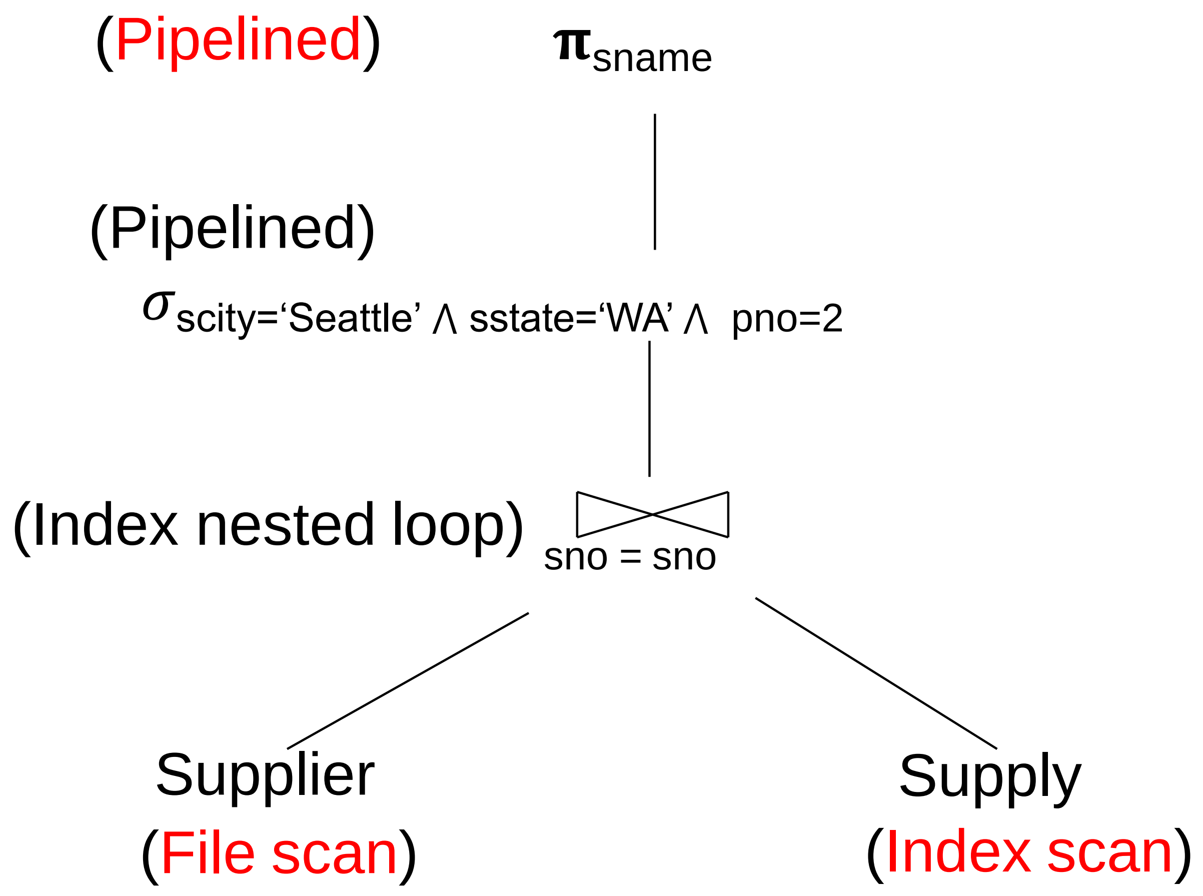
What We Also Know

- For each logical plan…
- There exist many physical plans

Example Query: Physical Plan 1



Example Query: Physical Plan 2

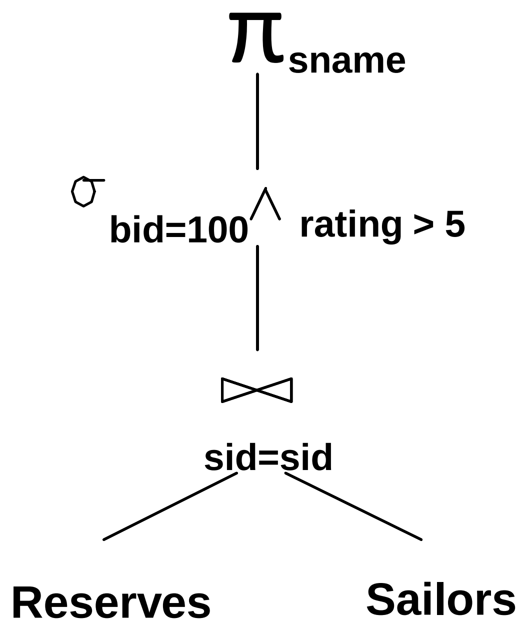


Query Optimization

1. Transformation produces relational algebra expression per "block"
2. Then, for each block, several alternative query plans are considered
3. Plan with lowest estimated cost is selected

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

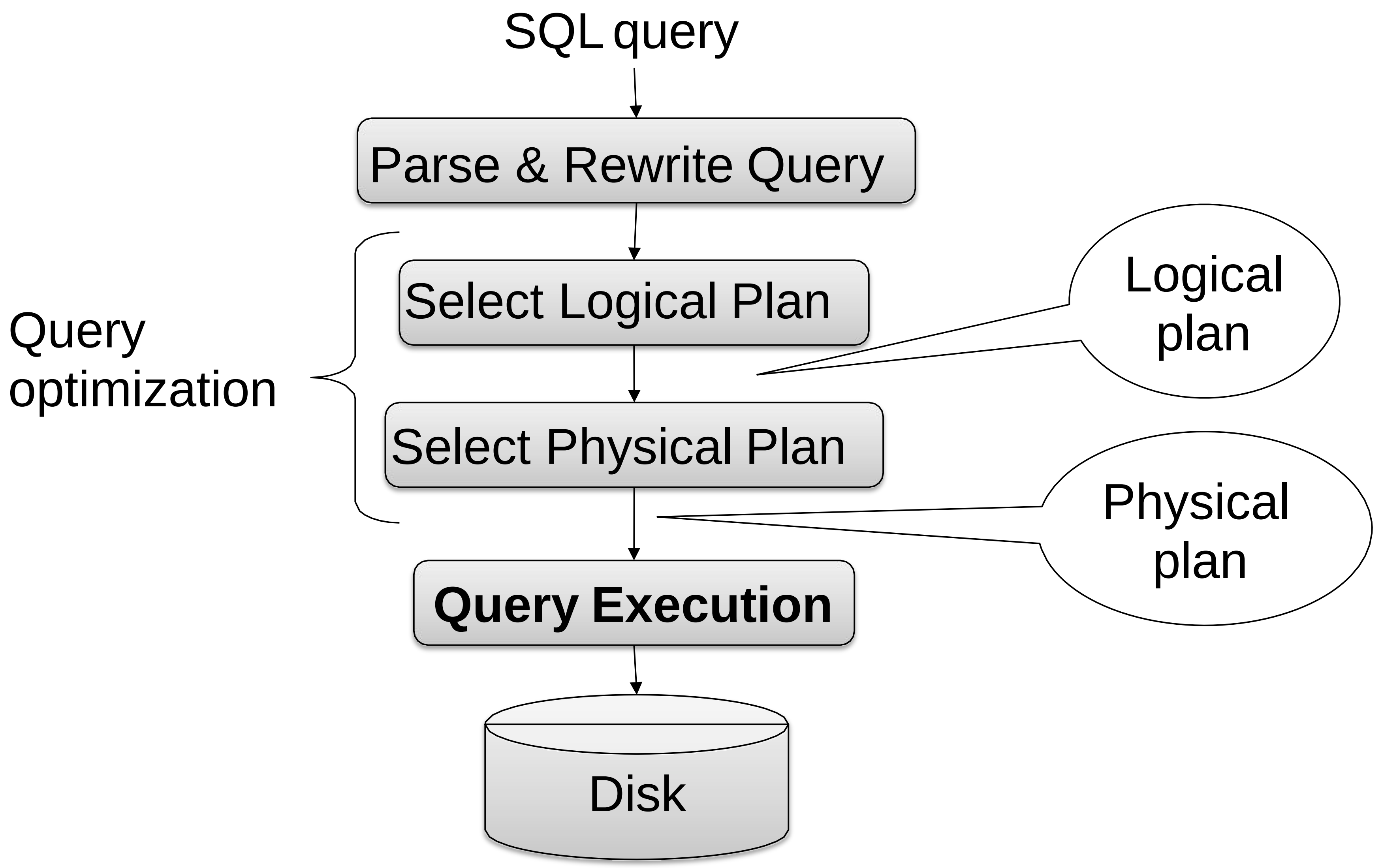
$$\pi_{(sname)}\sigma_{(bid=100 \wedge rating > 5)}\ (\text{Reserves} \bowtie \text{Sailors})$$



Query Optimizer Overview

- **Input**: A logical query plan
- **Output**: A good physical query plan
- **Basic query optimization algorithm**
  - Enumerate alternative plans (logical and physical)
  - Compute estimated cost of each plan
    - Compute number of I/Os
    - Optionally take into account other resources
  - Choose plan with lowest cost
  - This is called cost-based optimization

Steps in Query Processing



Query Execution Models

- A DBMS's processing model defines how the system executes a query plan.
  - Different trade-offs for workloads

- Approach #1: Interpreted execution
- Approach #2: Compiled execution

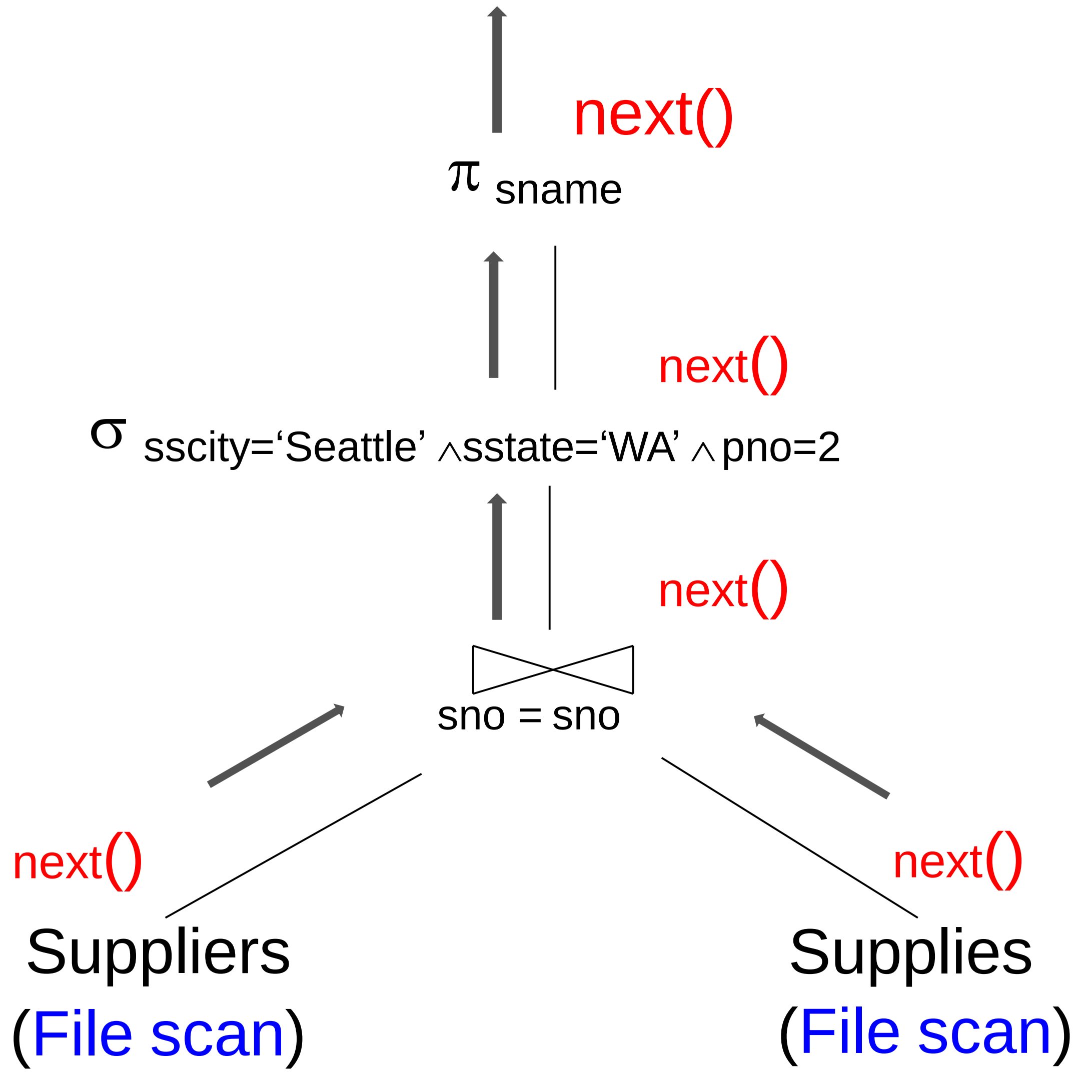
Intepreted execution with Volcano model

Each operator implements an iterator interface

- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **next()**
  - Operator invokes get_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**: clean-up state

Pipelined Execution

Tuples generated by an operator are immediately sent to the parent

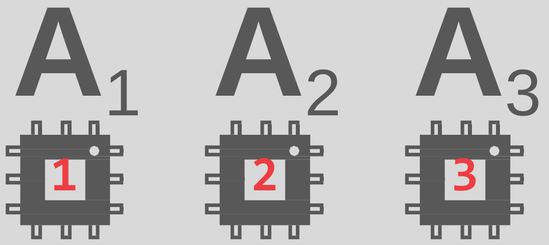


Pipelined Execution

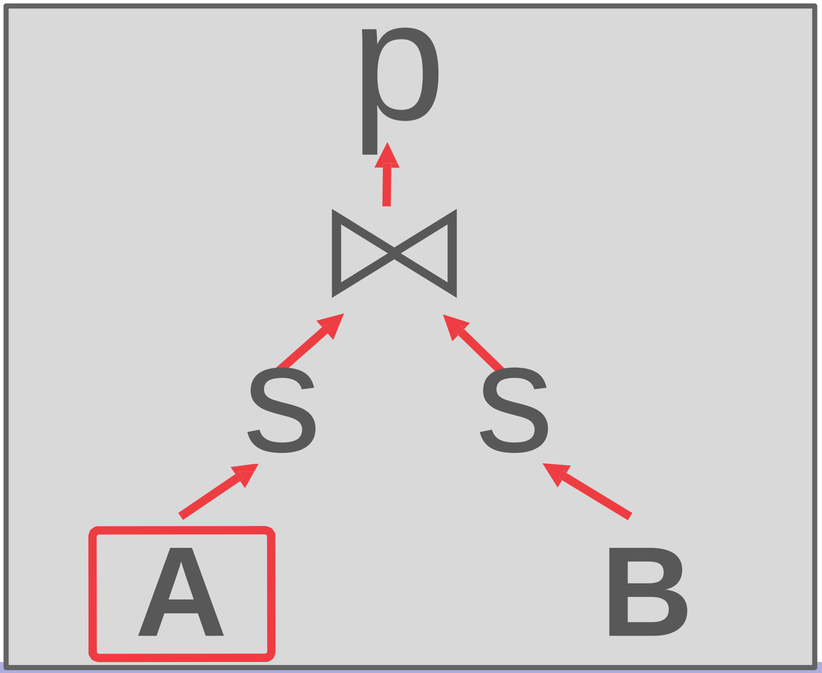
- Tuples generated by an operator are immediately sent to the parent

- Benefits:
  - Pull based: No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Enables implementation of parallelization as an operator

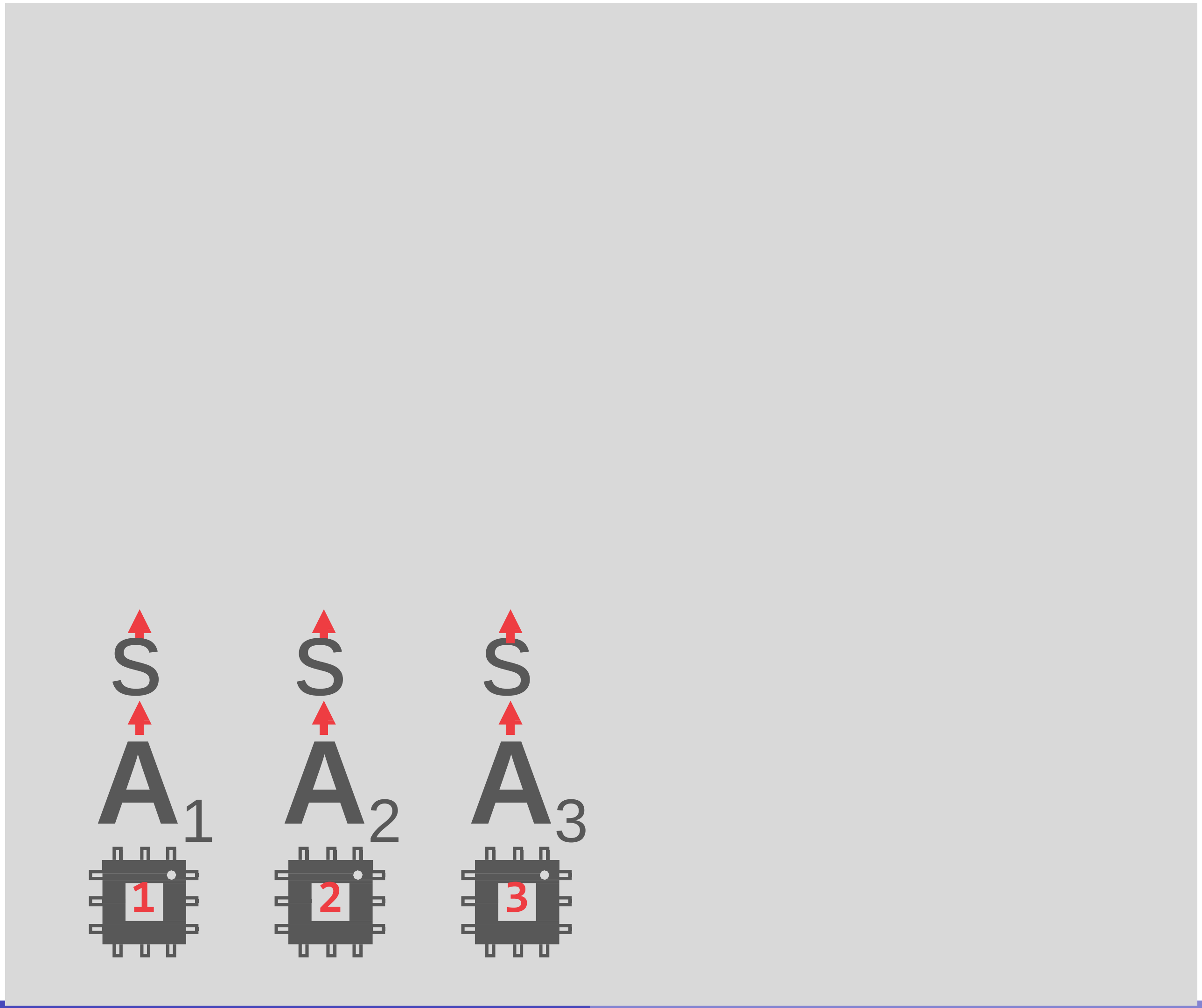
Exchange parallelization



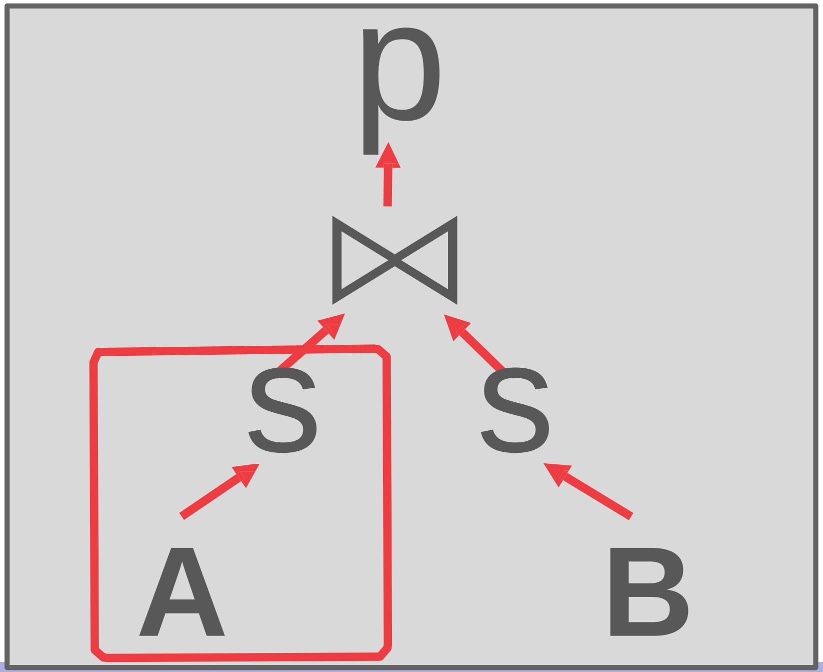
```
SELECT A.id, B.value
  FROM A JOIN B
     ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



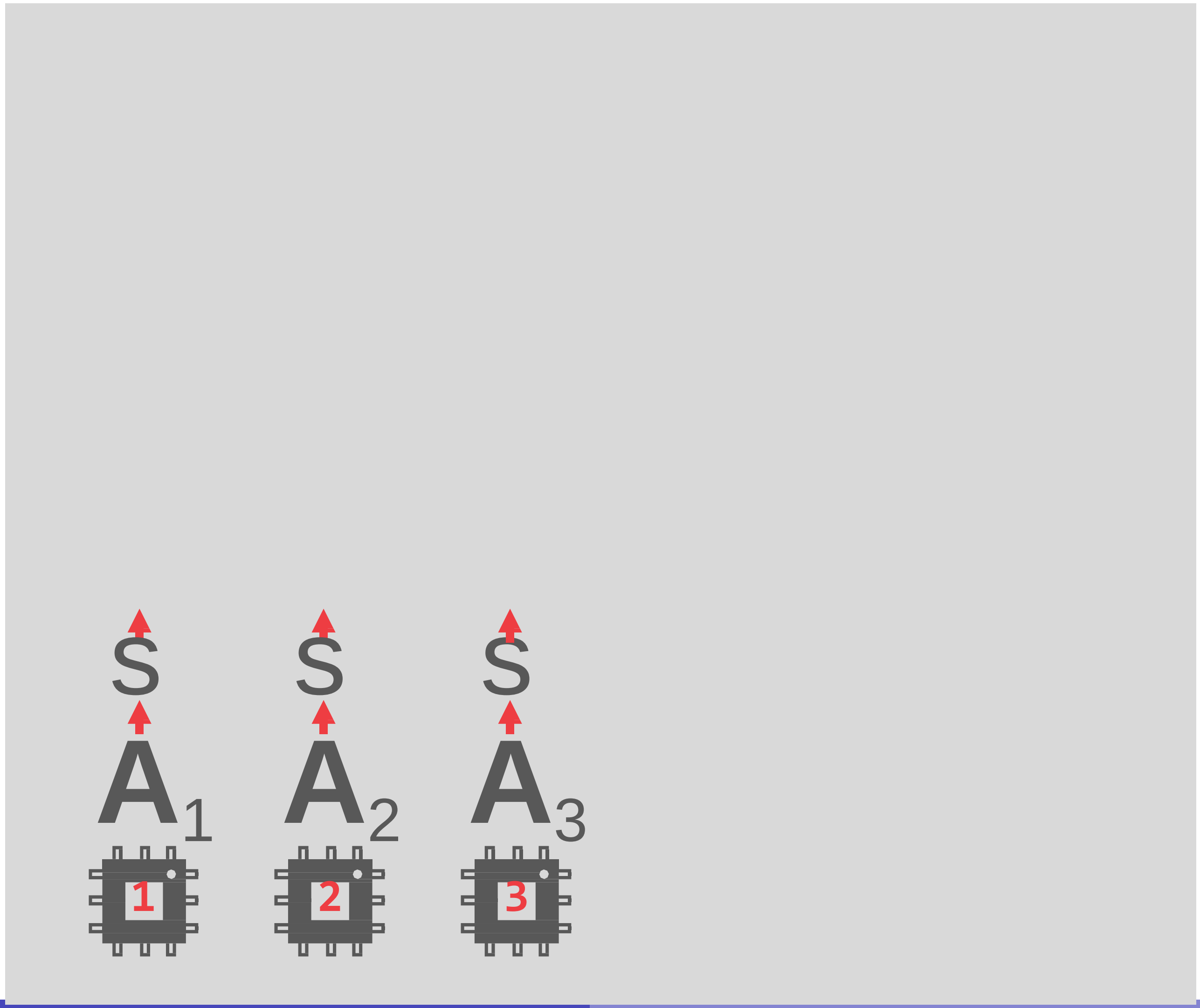
Exchange parallelization



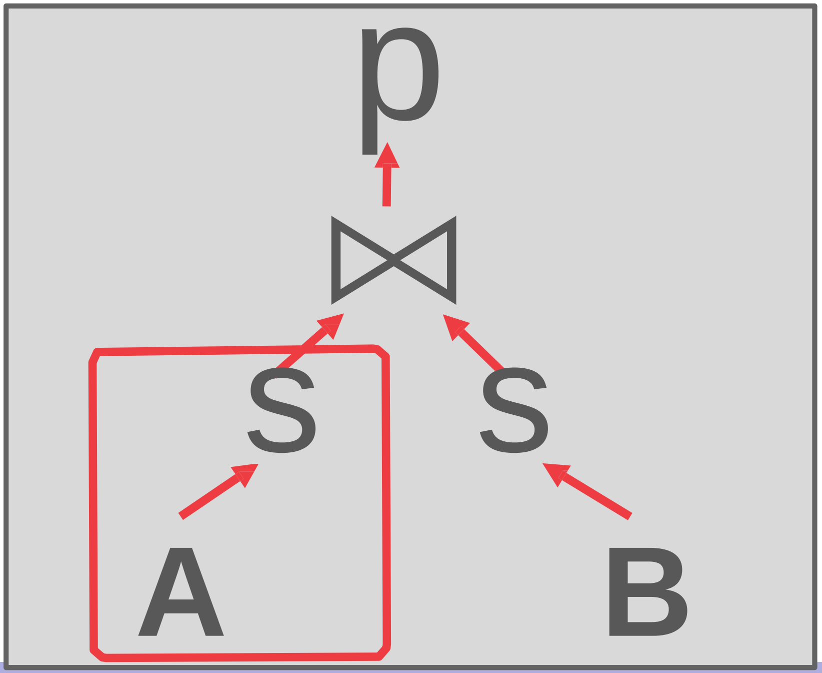
```
SELECT A.id, B.value
  FROM A JOIN B
     ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



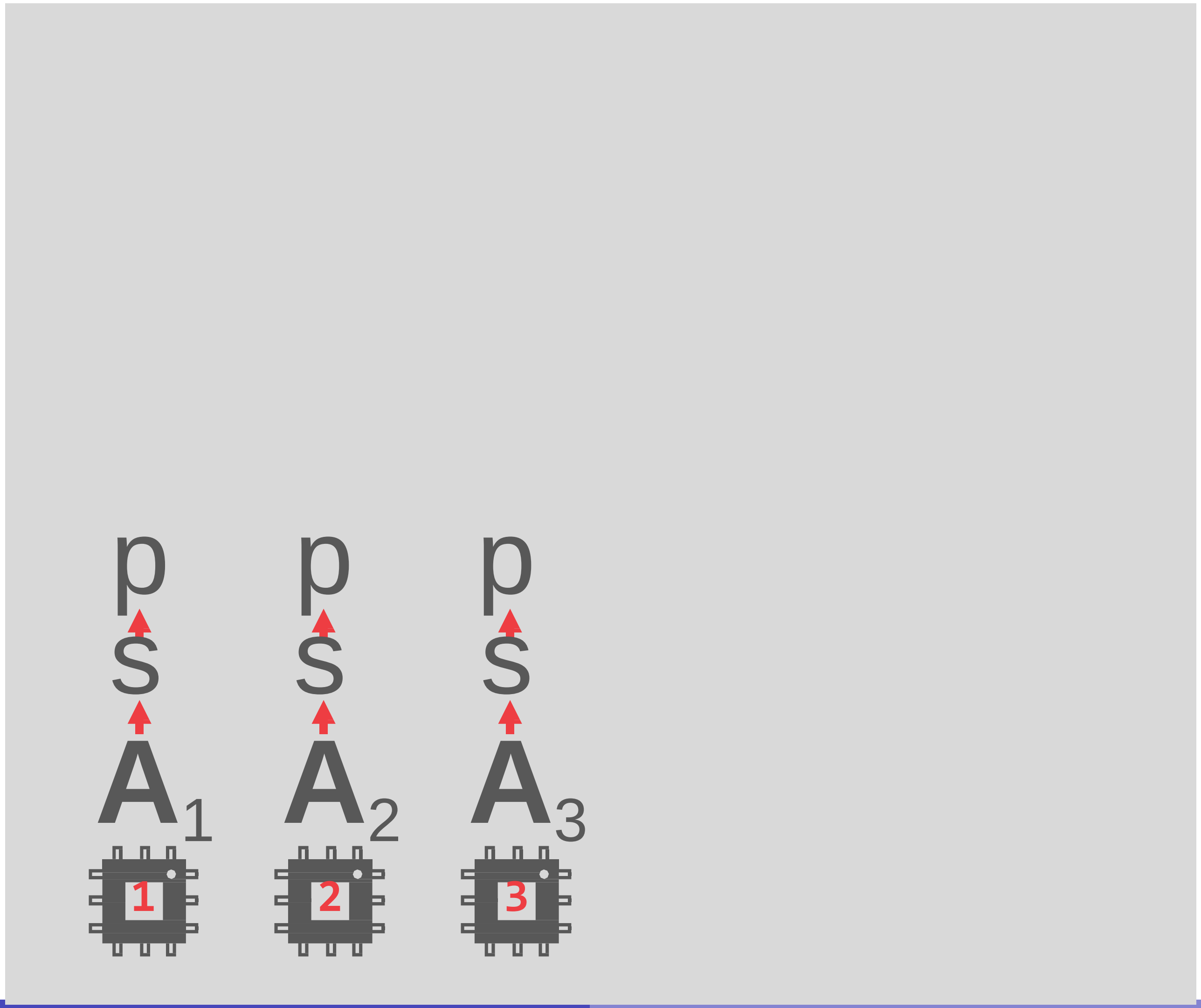
Exchange parallelization



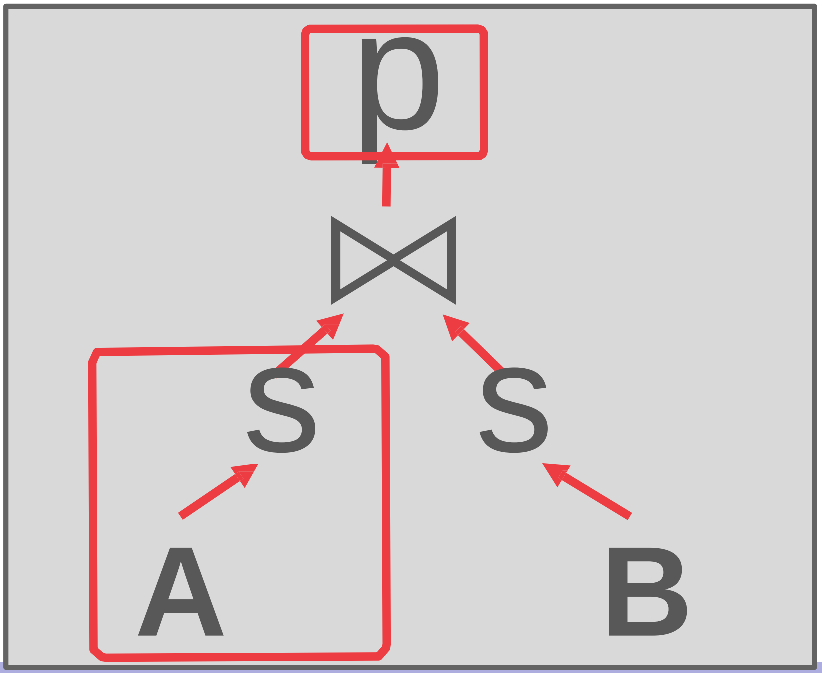
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



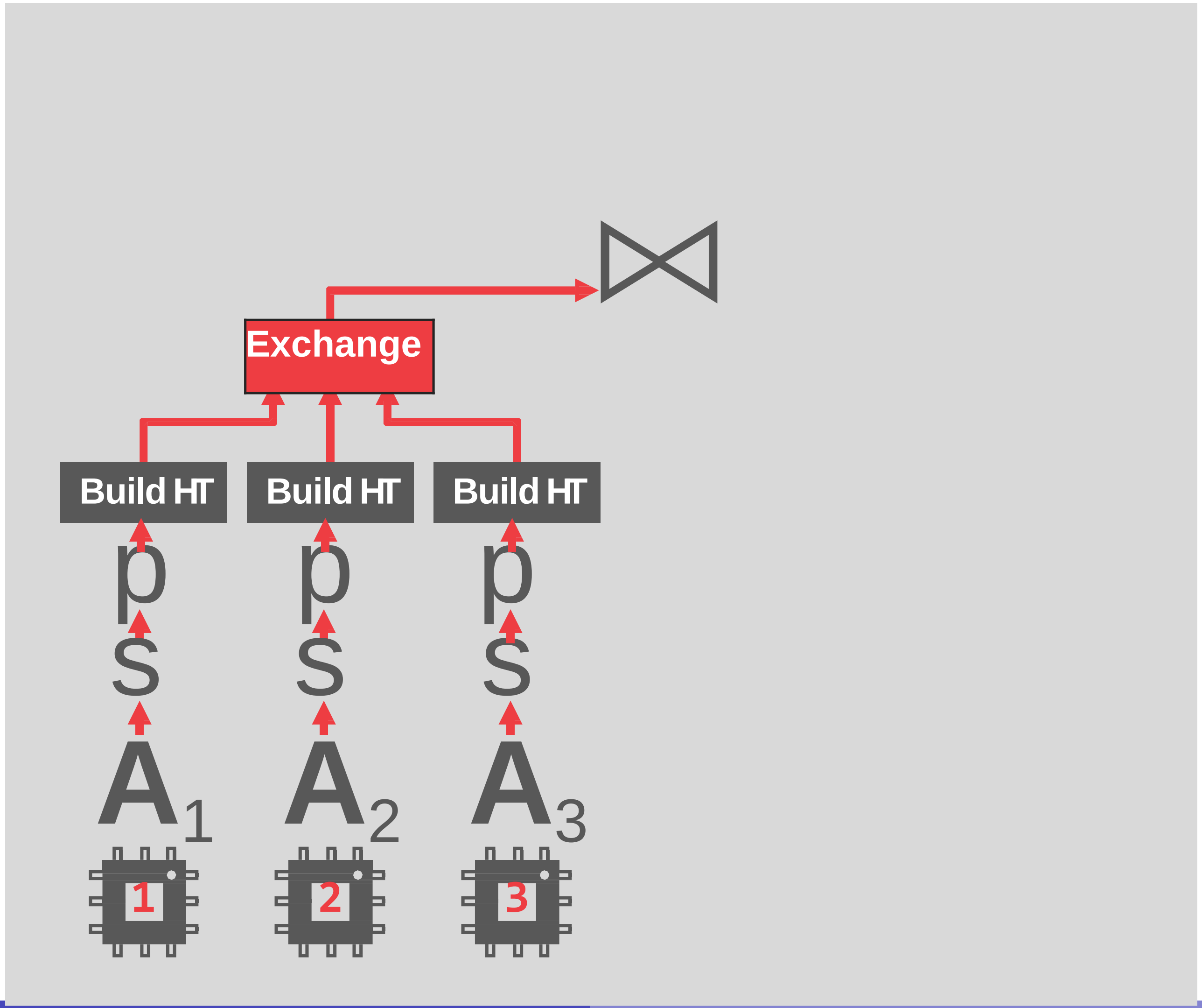
Exchange parallelization



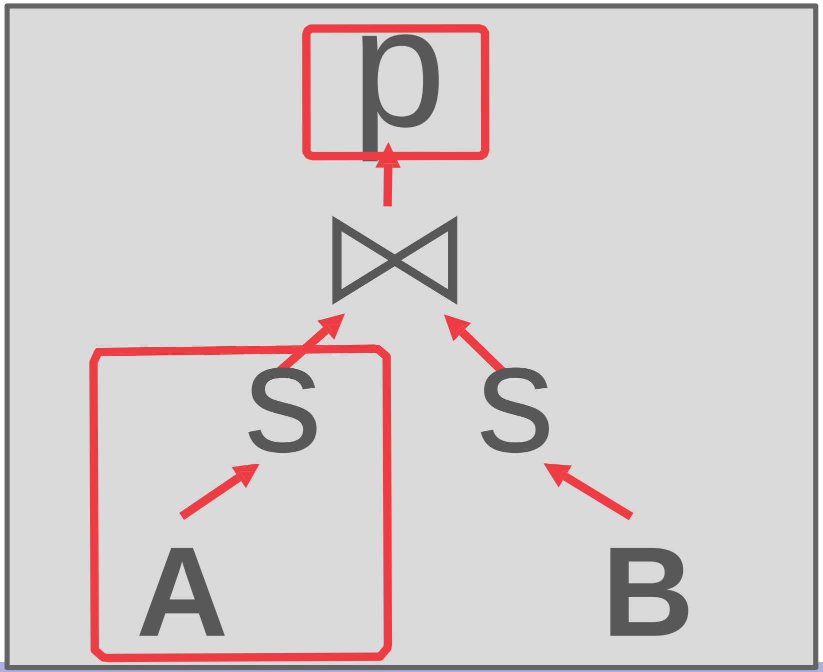
```
SELECT A.id, B.value
   FROM A JOIN B
        ON A.id = B.id
  WHERE A.value < 99
     AND B.value > 100
```



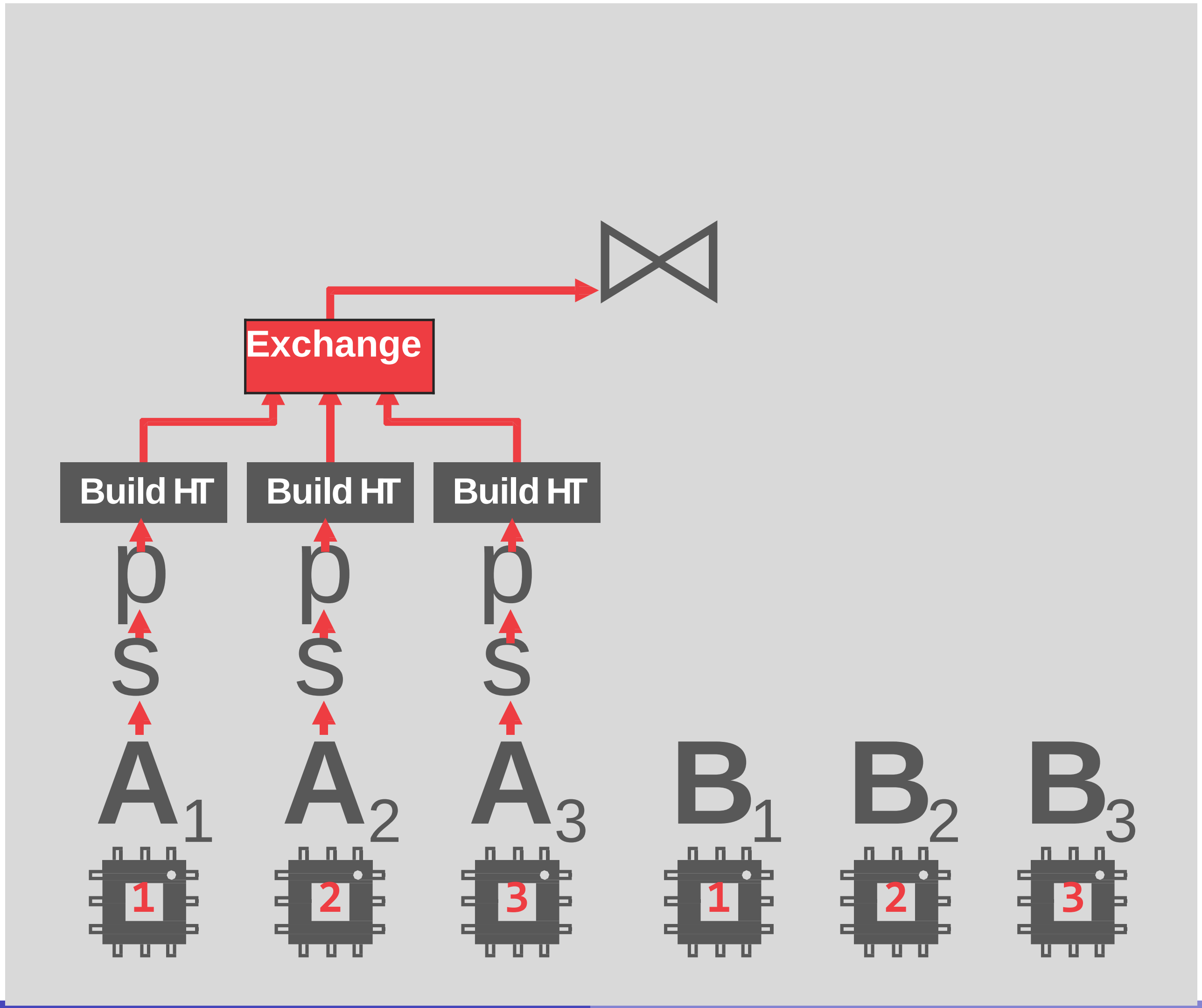
Exchange parallelization



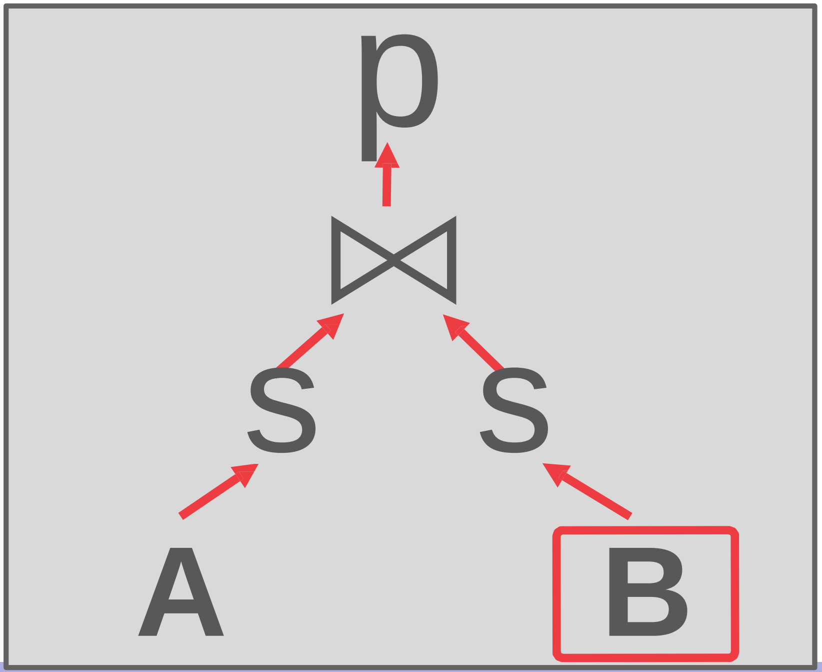
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



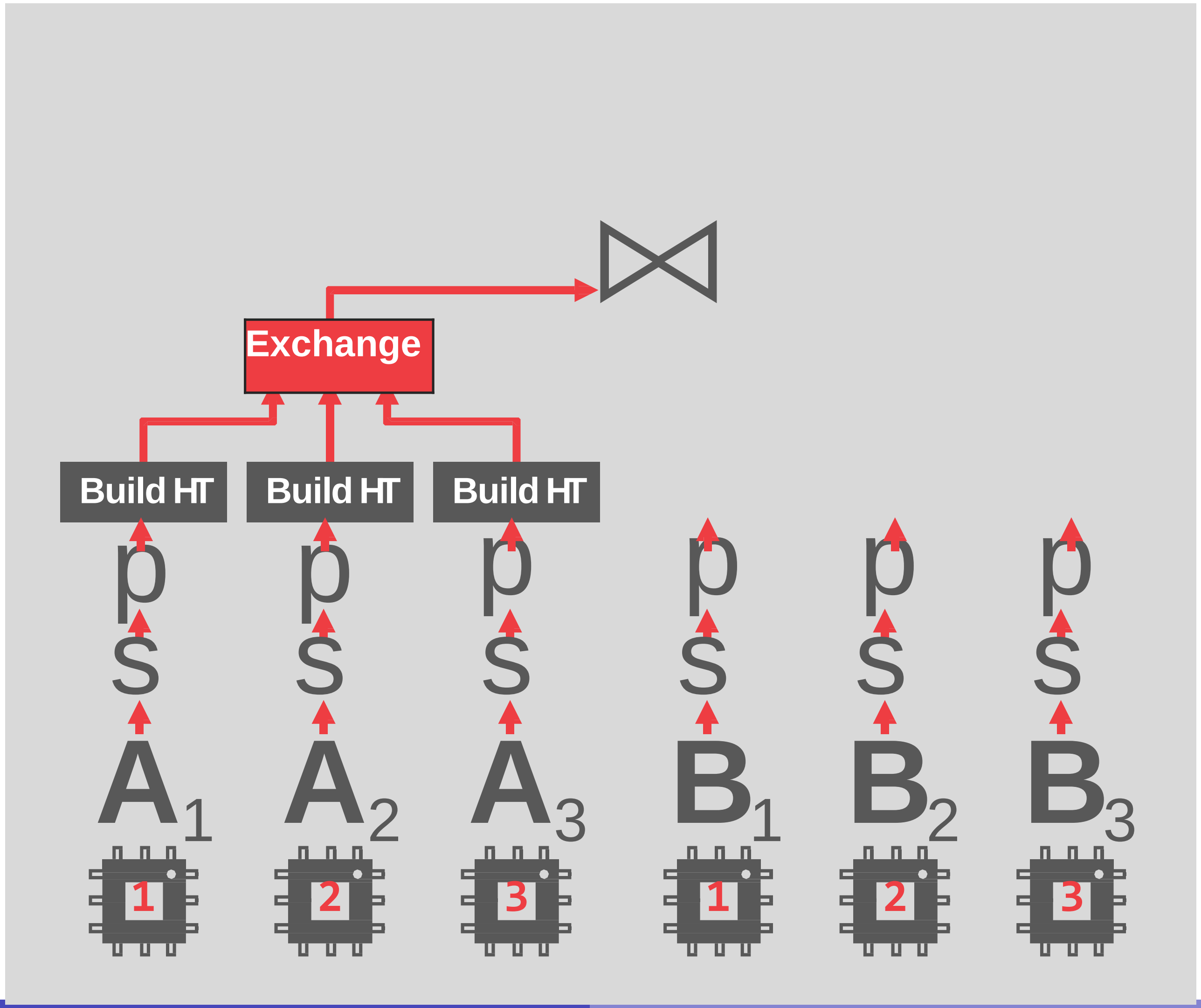
Exchange parallelization



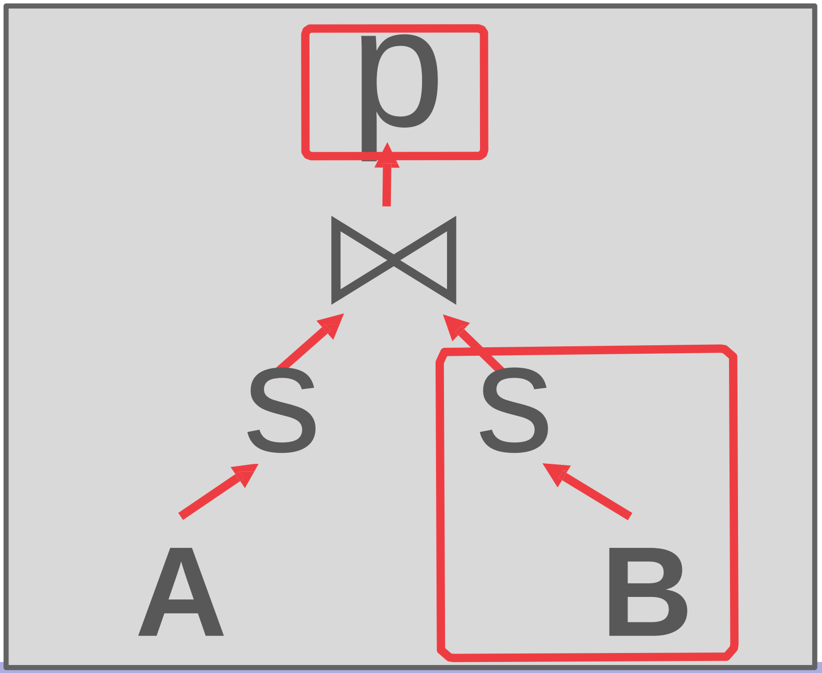
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



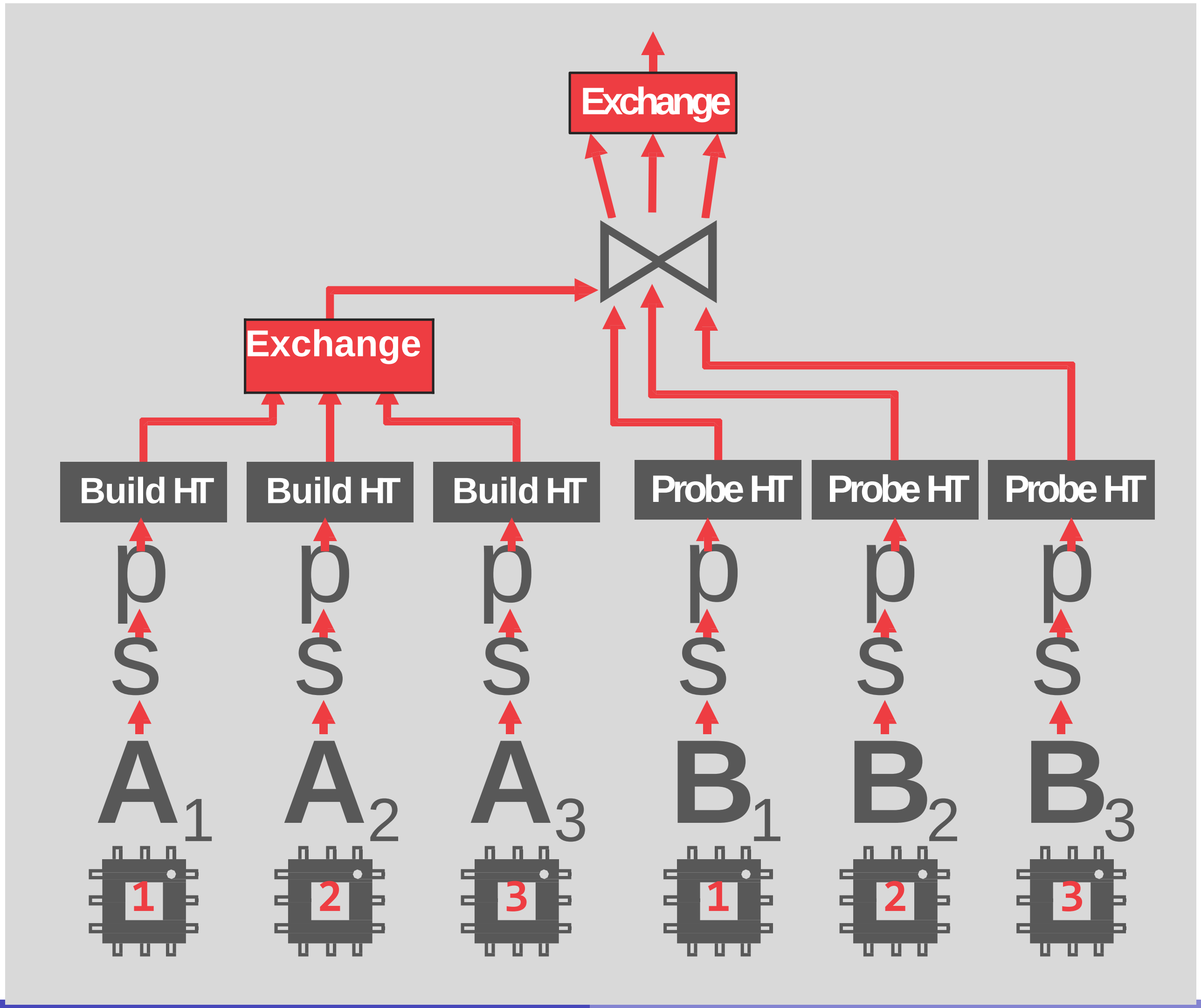
Exchange parallelization



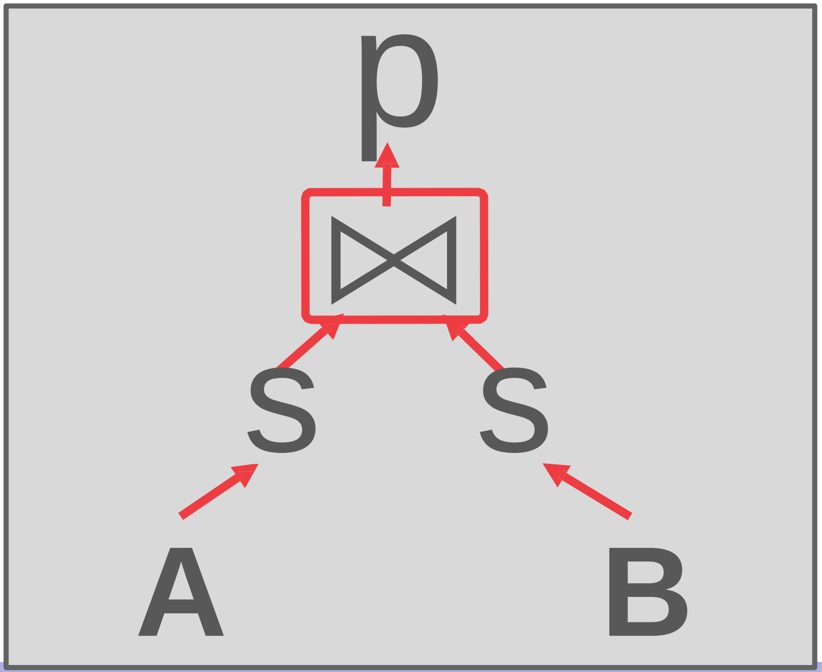
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



Exchange parallelization



```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



Pipelined Execution

- Tuples generated by an operator are immediately sent to the parent
- Benefits:
  - Pull based: No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Enables implementation of parallelization as an operator
- Drawback
  - High function call overhead
  - Difficult to perform SIMD-style vectorization

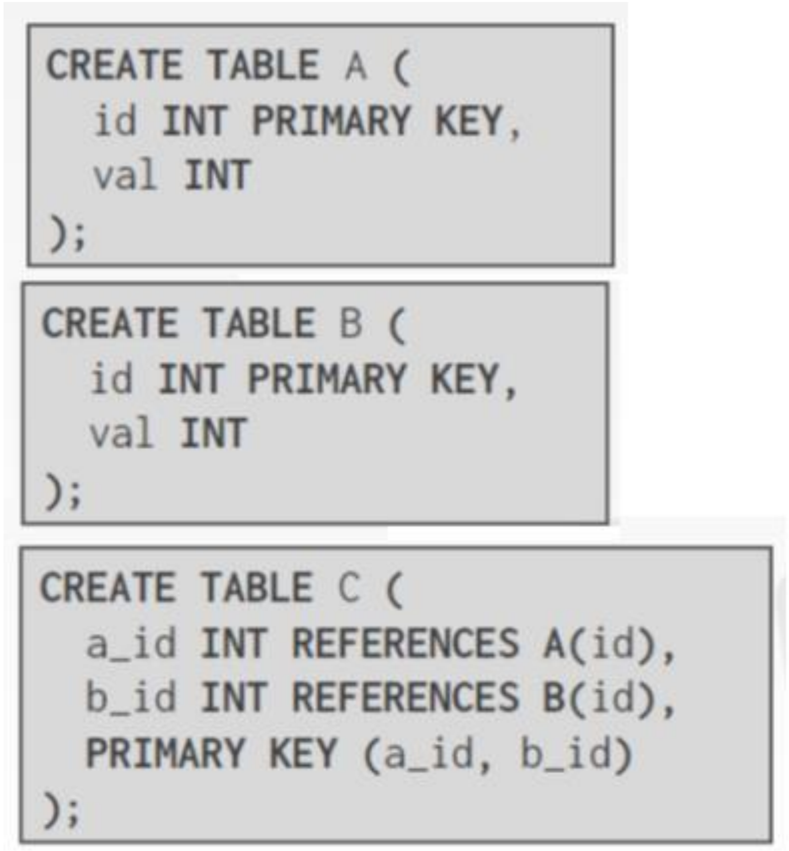
Compiled execution

- Compile queries in-memory into native code

- Organizes query processing in a way to keep a tuple in CPU registers for as long as possible.
  - Push-based vs. Pull-based
  - Data Centric vs. Operator Centric

- LLVM typically used for compilation
  - Collection of modular and reusable compiler and toolchain technologies.
  - Core component is a low-level programming language (IR) that is like assembly.

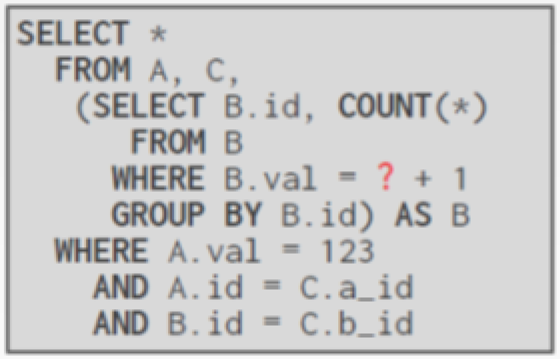
Query interpretation vs compilation: Example



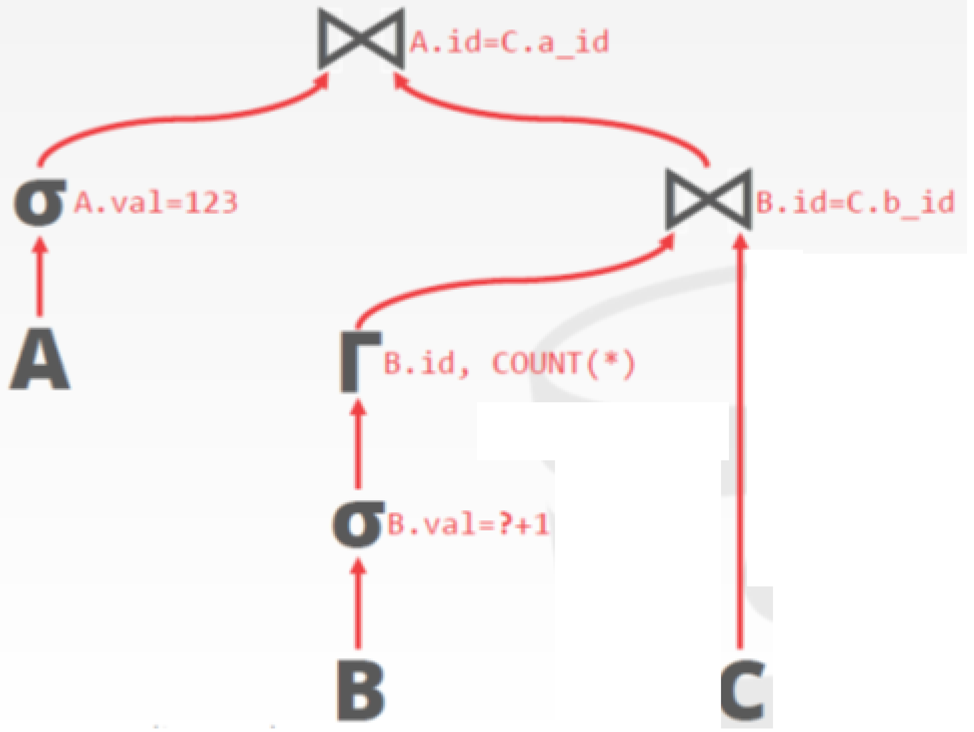
```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT
);
```

```
CREATE TABLE B (
  id INT PRIMARY KEY,
  val INT
);
```

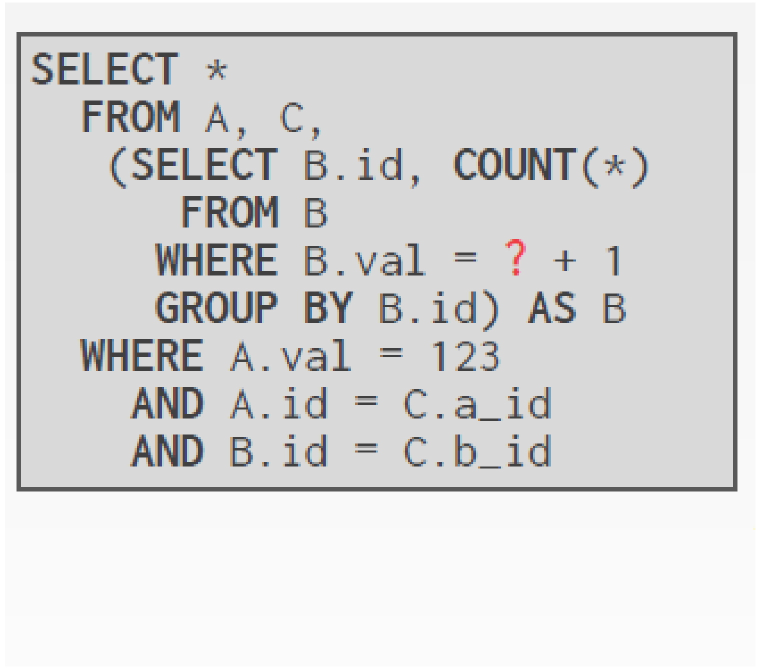
```
CREATE TABLE C (
  a_id INT REFERENCES A(id),
  b_id INT REFERENCES B(id),
  PRIMARY KEY (a_id, b_id)
);
```



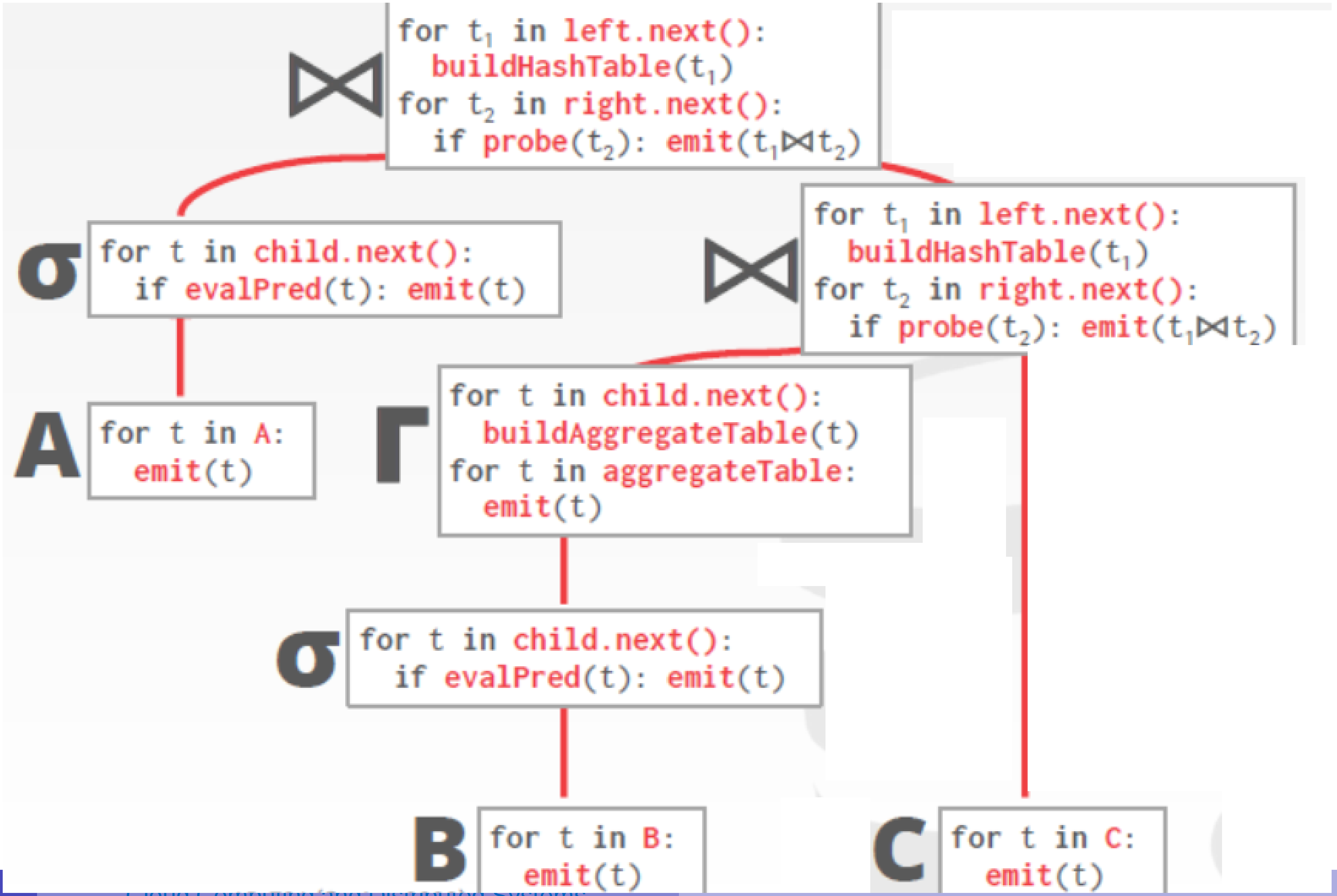
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



Query interpretation vs compilation: Example



```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



⋈

```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

σ

```
for t in child.next():
  if evalPred(t): emit(t)
```

⋈

```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

A

```
for t in A:
  emit(t)
```

Γ

```
for t in child.next():
  buildAggregateTable(t)
for t in aggregateTable:
  emit(t)
```

σ

```
for t in child.next():
  if evalPred(t): emit(t)
```

B

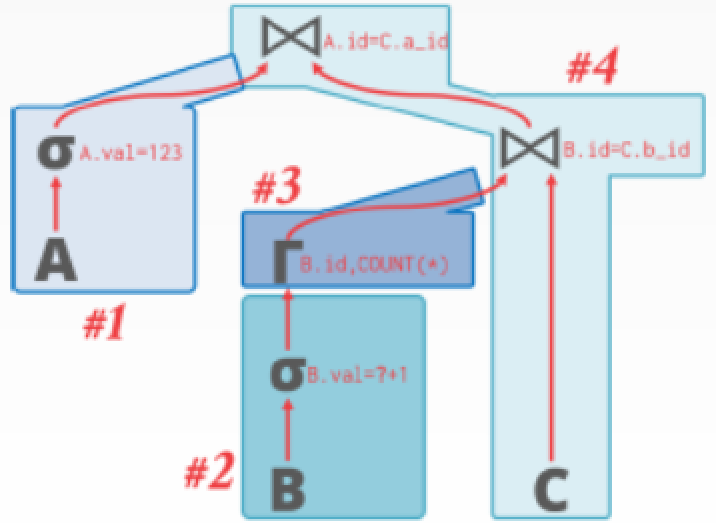
```
for t in B:
  emit(t)
```

C

```
for t in C:
  emit(t)
```

Query interpretation vs compilation: Example

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
   AND A.id = C.a_id
   AND B.id = C.b_id
```



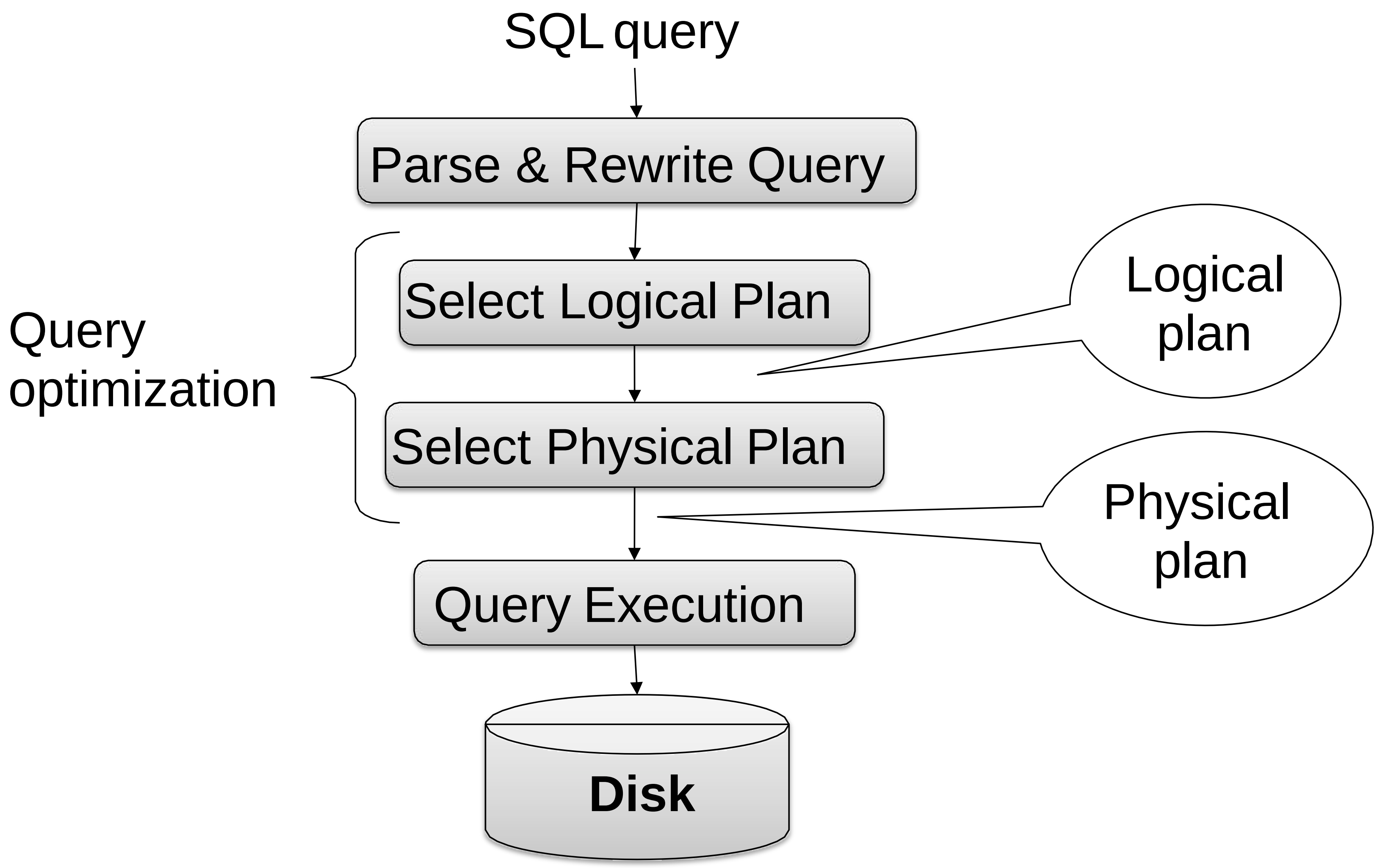
```
#1  for t in A:
      if t.val == 123:
        Materialize t in HashTable ⋈(A.id=C.a_id)

#2  for t in B:
      if t.val == <param> + 1:
        Aggregate t in HashTable Γ(B.id)

#3  for t in Γ(B.id):
      Materialize t in HashTable ⋈(B.id=C.b_id)

#4  for t3 in C:
      for t2 in ⋈(B.id=C.b_id):
        for t1 in ⋈(A.id=C.a_id):
          emit(t1⋈t2⋈t3)
```

On-disk Storage



DBMS on-disk storage

- The DBMS stores a database as one or more files on disk typically in a proprietary format.
  - The OS doesn't know anything about the contents of these files.

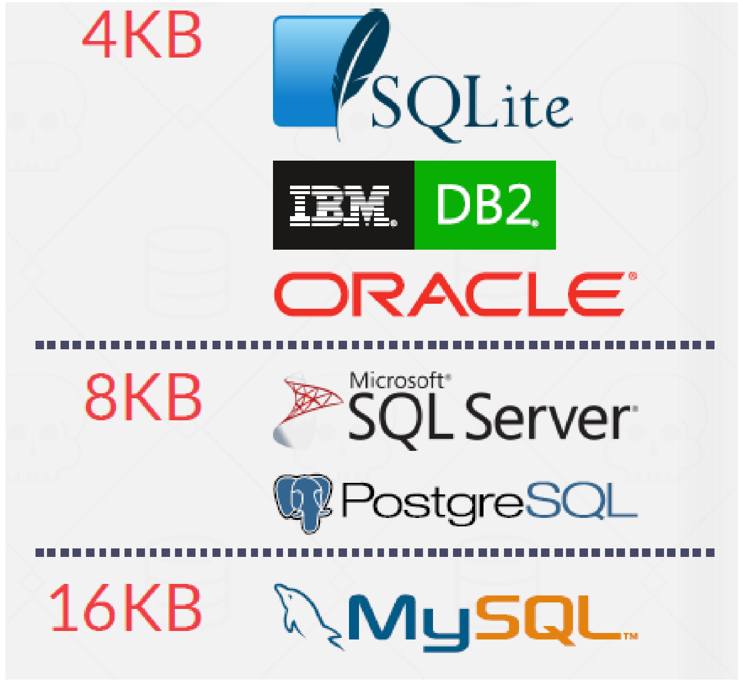
- DBMS storage manager is responsible for maintaining a database's files
  - It organizes the files as a collection of pages.
  - Tracks data read/written to pages.
  - Tracks the available space.

DBMS Page

- A page is a fixed-size block of data.
  - It can contain tuples, meta-data, indexes, log records…

- There are three different notions of "pages" in a DBMS:
  - Hardware Page (usually 4KB)
  - OS Page (usually 4KB)
  - Database Page (512B-16KB)

- We wont talk about page organization
  - Heap/tree/sorted/hashing files
  - Pages organiz



Storage Models

- The relational model does not specify how a DBMS must store all a tuple in a page.

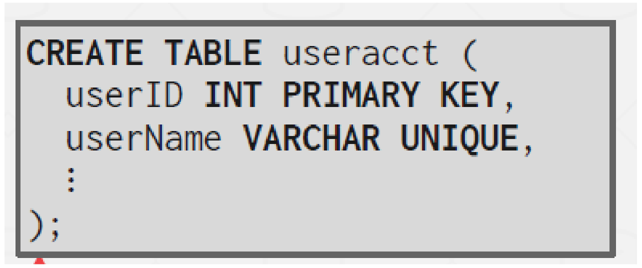
- Storage models
  - Dictate how tuples are stored within a page
  - Different models optimized for different workloads

Database Workloads

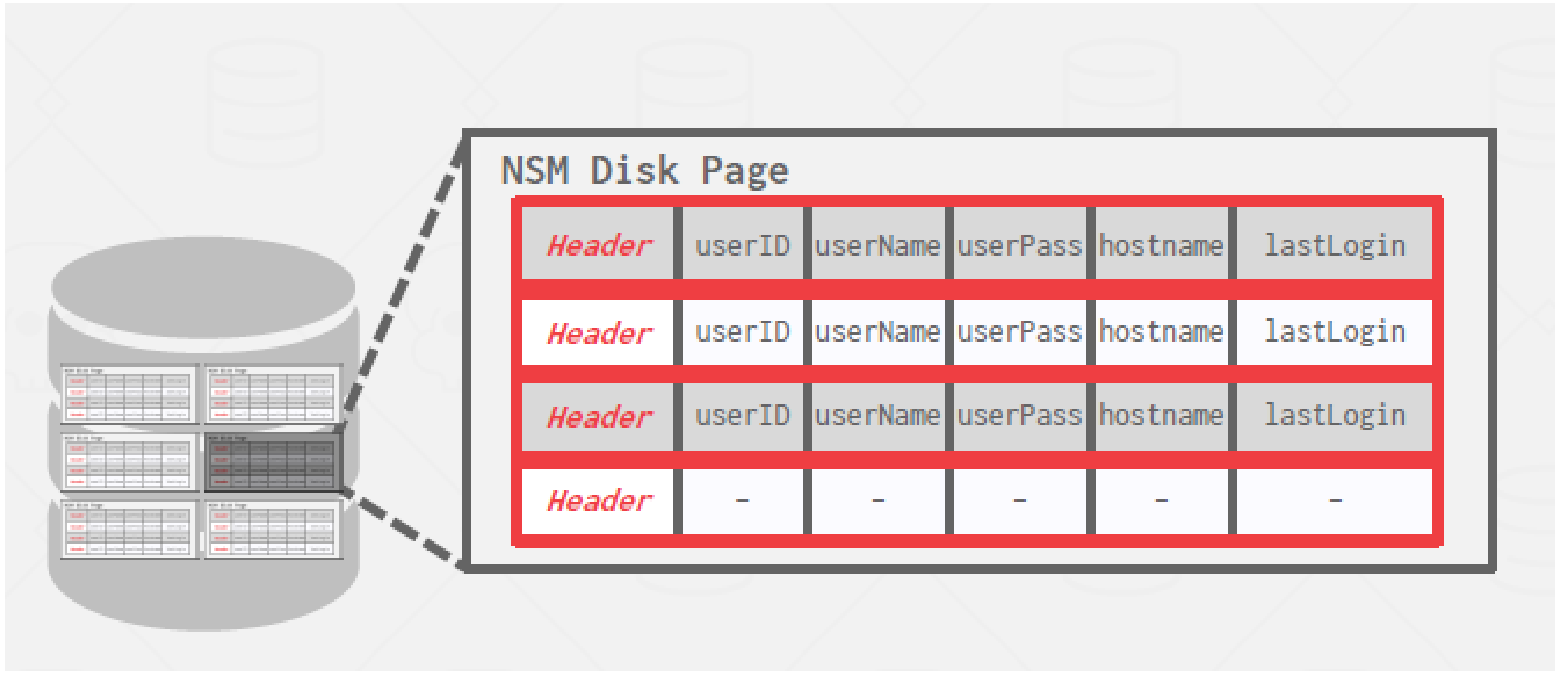
- **On-Line Transaction Processing (OLTP)**
  - Fast operations that only read/update a small amount of data each time.
- **On-Line Analytical Processing (OLAP)**
  - Complex queries that read a lot of data to compute aggregates.
- **Hybrid Transaction + Analytical Processing**
  - OLTP + OLAP together on the same database instance

N-ary storage model

- DBMS stores all attributes for a single tuple contiguously in a page.

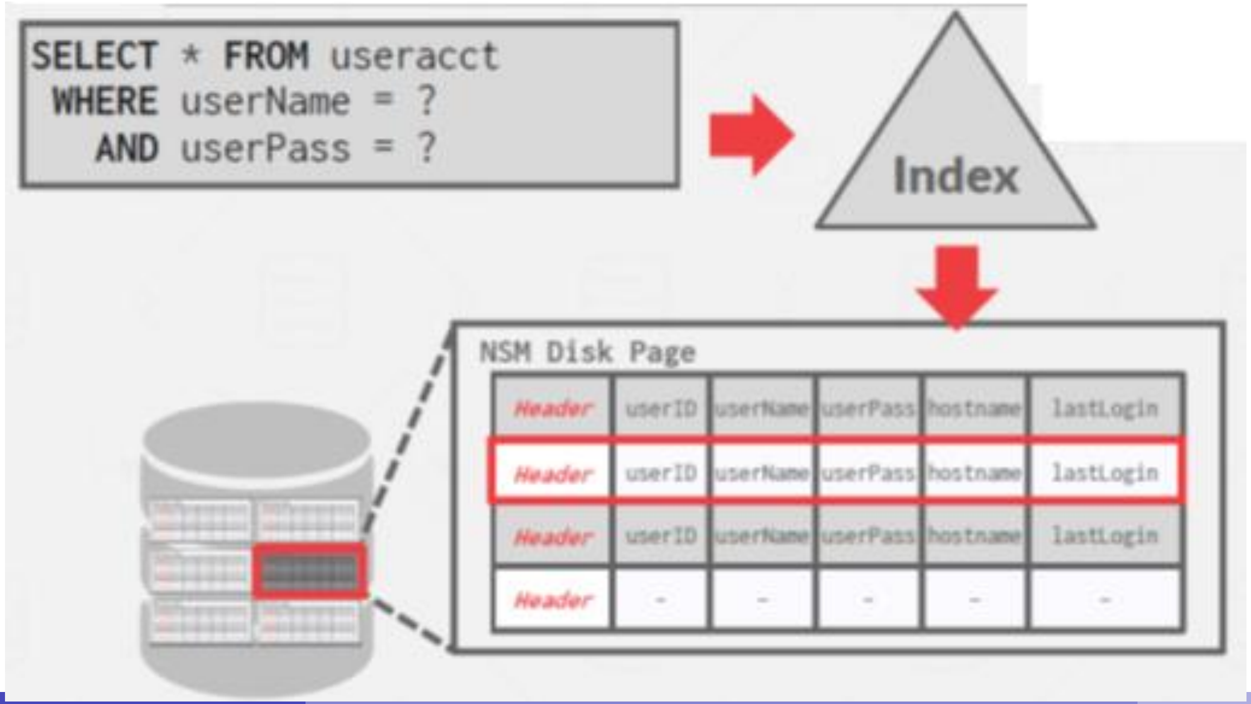


```
CREATE TABLE useracct (
  userID INT PRIMARY KEY,
  userName VARCHAR UNIQUE,
  ⋮
);
```



N-ary storage model

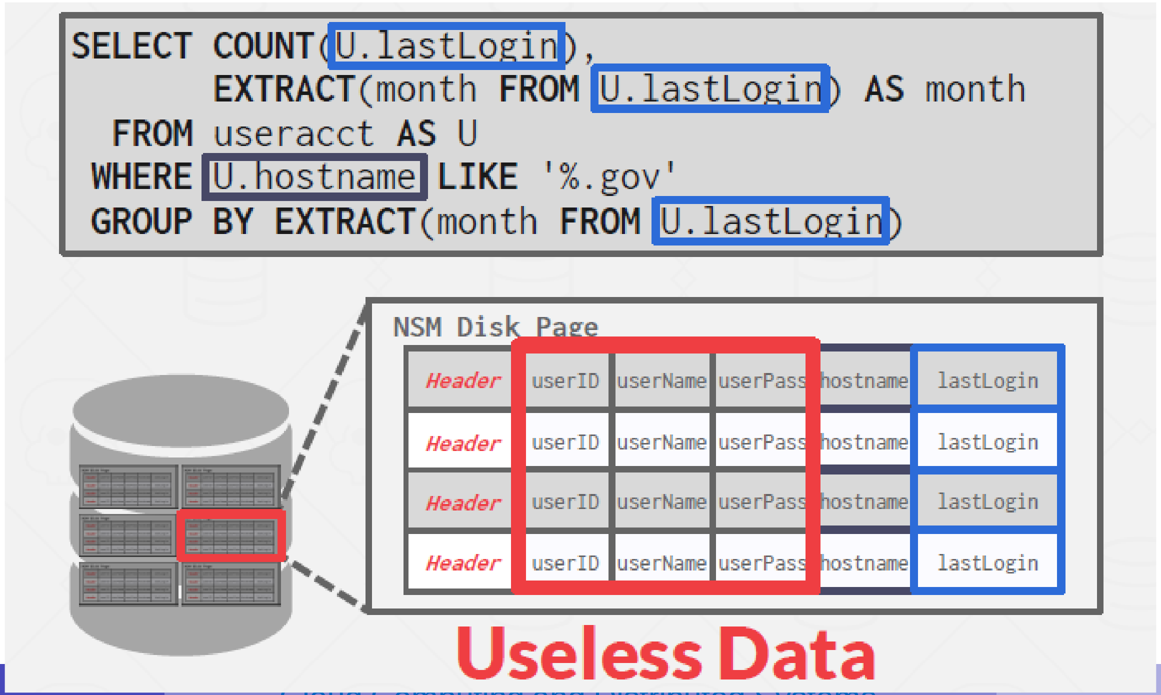
- The DBMS stores all attributes for a single tuple contiguously in a page.
- Ideal for OLTP workloads where queries tend to operate only on an individual entity and insert heavy workloads.



```
SELECT * FROM useracct
 WHERE userName = ?
   AND userPass = ?
```

N-ary storage model

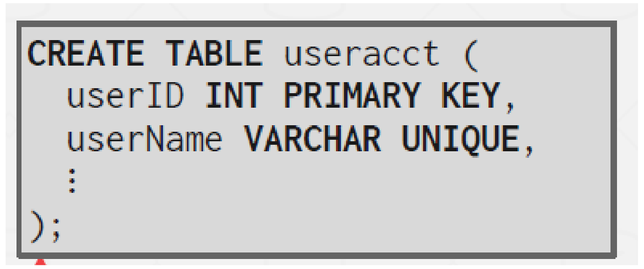
- **Advantages**
  - Fast inserts, updates, and deletes.
  - Good for queries that need the entire tuple.
- **Disadvantages**
  - Not good for scanning large portions of the table and/or a subset of the attributes.



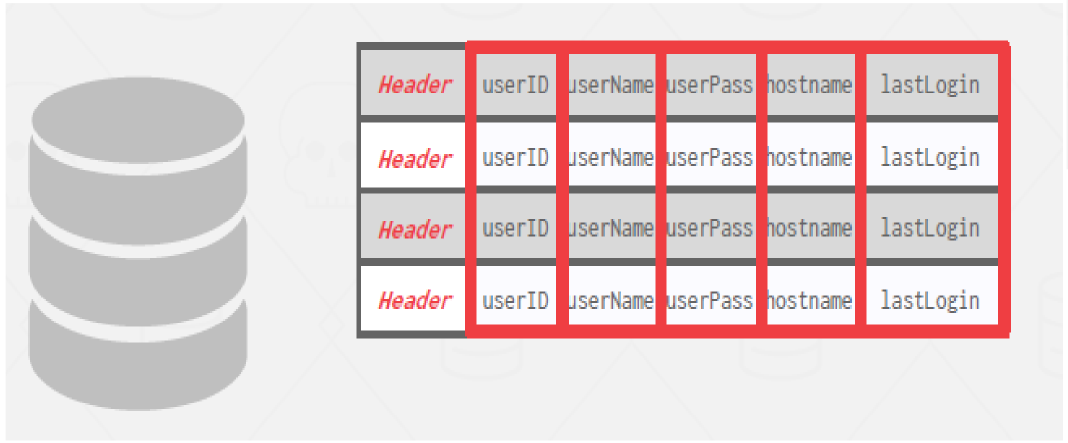
```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

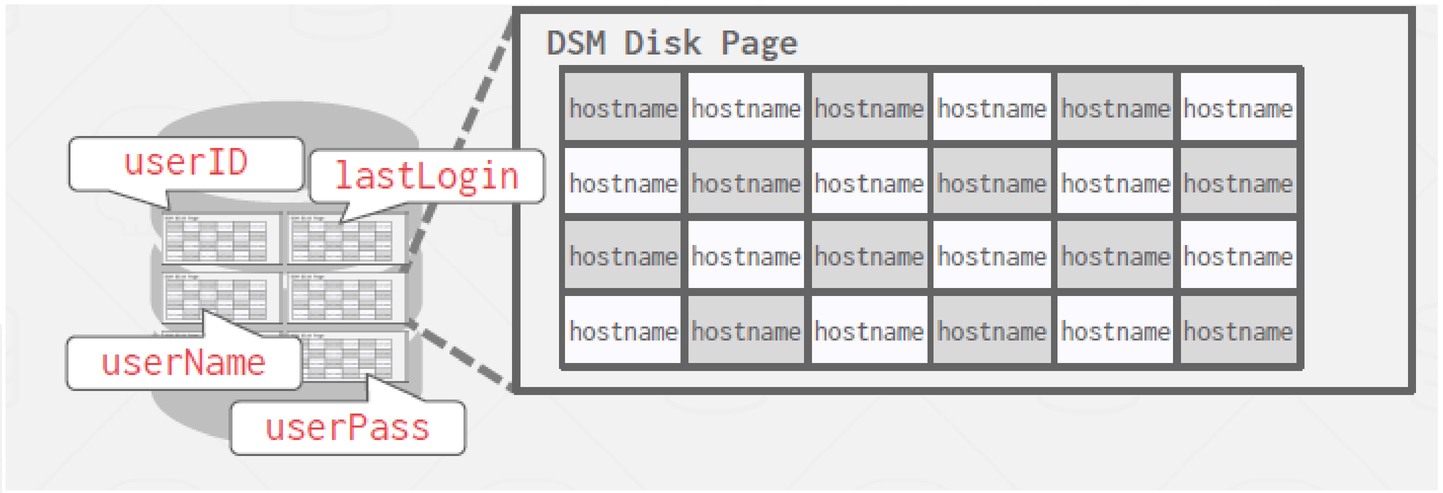
Decomposition storage model (DSM)

- The DBMS stores the values of a single attribute for all tuples contiguously in a page.
- Also known as a "column store"



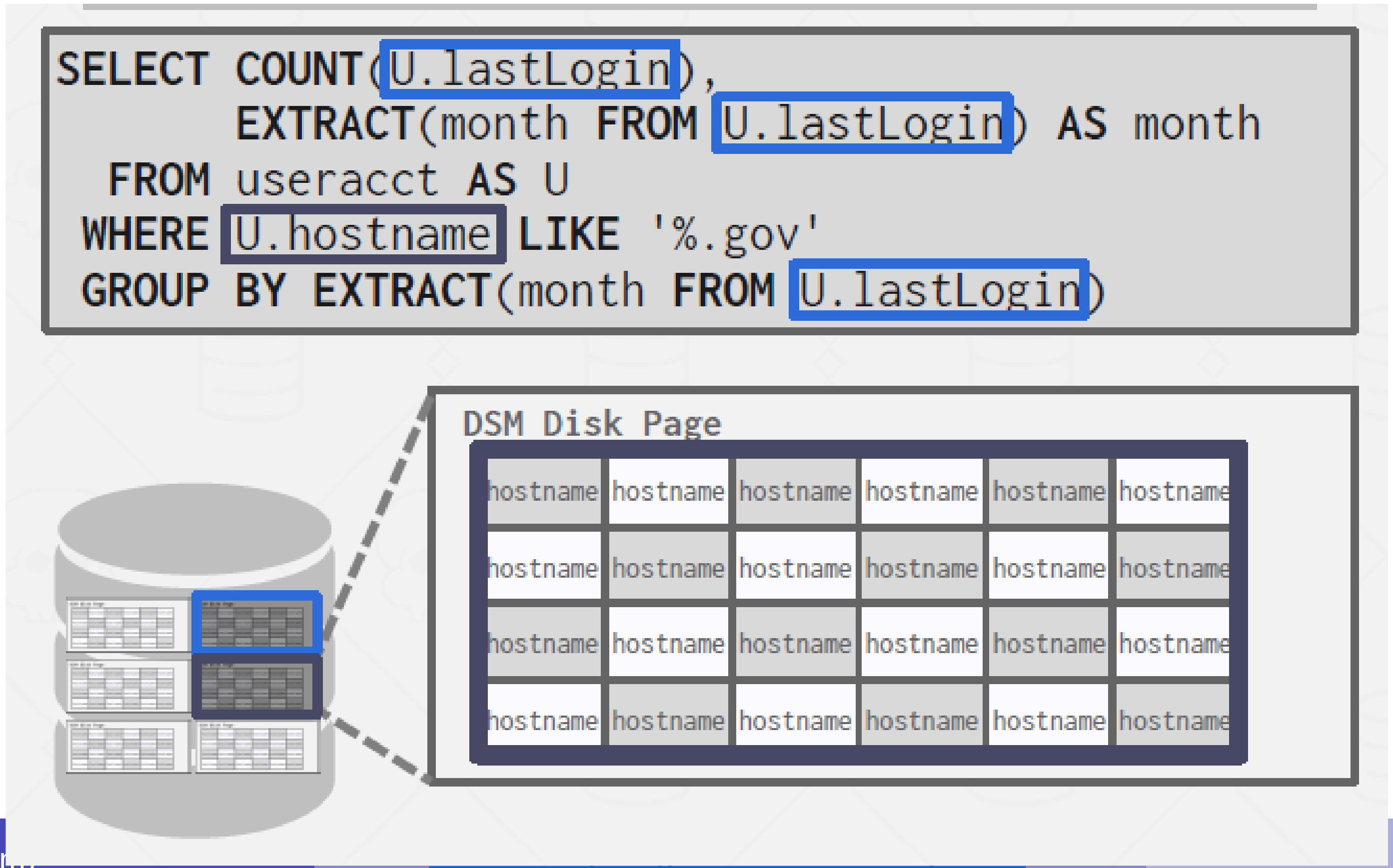
```
CREATE TABLE useracct (
  userID INT PRIMARY KEY,
  userName VARCHAR UNIQUE,
  ⋮
);
```





Decomposition storage model (DSM)

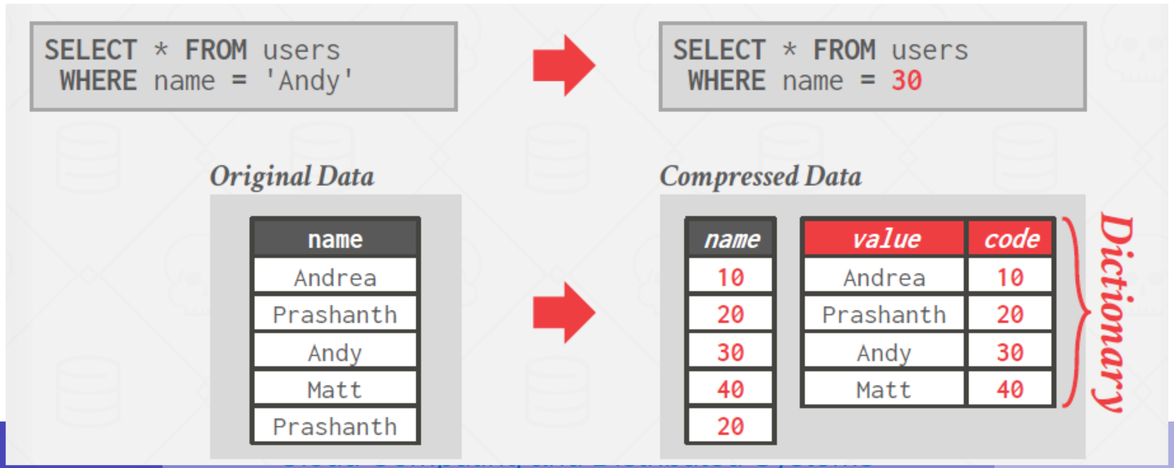
- Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes.



```
SELECT COUNT(U.lastLogin),
       EXTRACT(month FROM U.lastLogin) AS month
  FROM useracct AS U
 WHERE U.hostname LIKE '%.gov'
 GROUP BY EXTRACT(month FROM U.lastLogin)
```

Decomposition storage model (DSM)

- **Advantages**
  - Reduces the amount wasted I/O because the DBMS only reads the data that it needs.
  - Better query processing and data compression
- Dictionary compression example
  - Most used scheme in DBMSs enables queries over compressed data
  - Build a data structure that maps variable-length values to integer identifier.
  - Replace those values with their identifier in the dictionary data structure.



```
SELECT * FROM users
 WHERE name = 'Andy'
```

```
SELECT * FROM users
 WHERE name = 30
```

name
Andrea
Prashanth
Andy
Matt
Prashanth

name
10
20
30
40
20

value	code
Andrea	10
Prashanth	20
Andy	30
Matt	40

Storage Models: Summary

- N-ary storage model
  - **Advantages**
    - Fast inserts, updates, and deletes.
    - Good for queries that need the entire tuple.
  - **Disadvantages**
    - Not good for scanning large portions of the table and/or a subset of the attributes.

- Decomposition storage model
  - **Advantages**
    - Reduces the amount wasted I/O because the DBMS only reads the data that it needs.
    - Better query processing and data compression
  - **Disadvantages**
    - Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

It’s all about them tables

- Relational databases
  - Relational model: Logical data independence
  - Relational algebra: Algebraic optimization, declarative querying
  - Optimized access paths: Indexing, materialized views, …
  - Transactional semantics: ACID guarantees

- What is the link with MapReduce?

MR DBMS Comparison

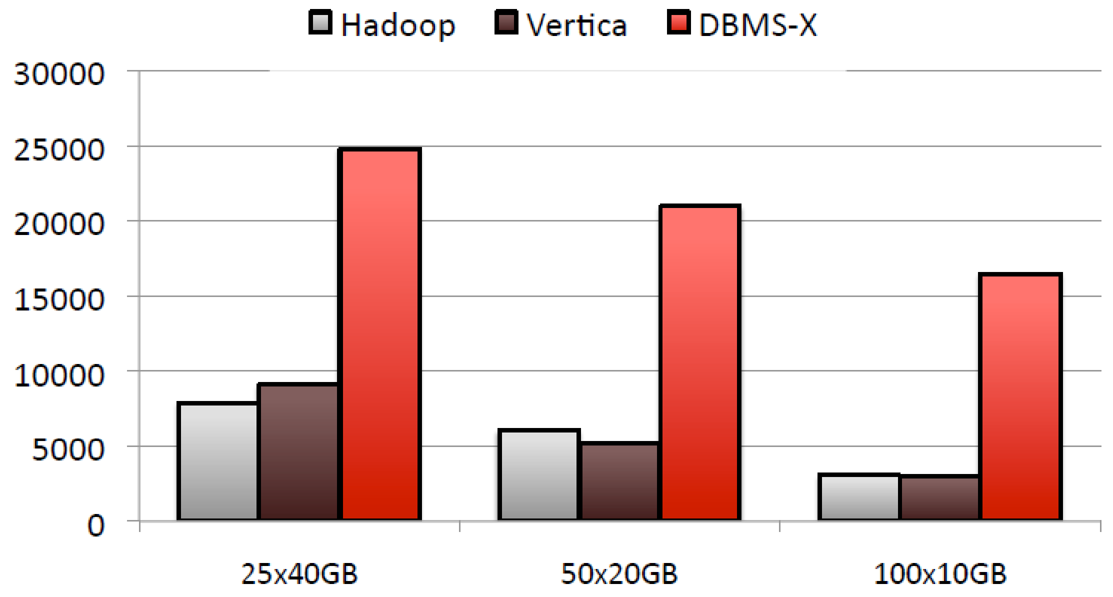
“MapReduce and Parallel DBMS: A comparison of approaches to large-scale data analysis” – Andy Pavlo ’09

- Tested Systems
  - Hadoop (MR)
  - Vertica (Columnar DBMS)
  - DBMS-X (Rowstore)

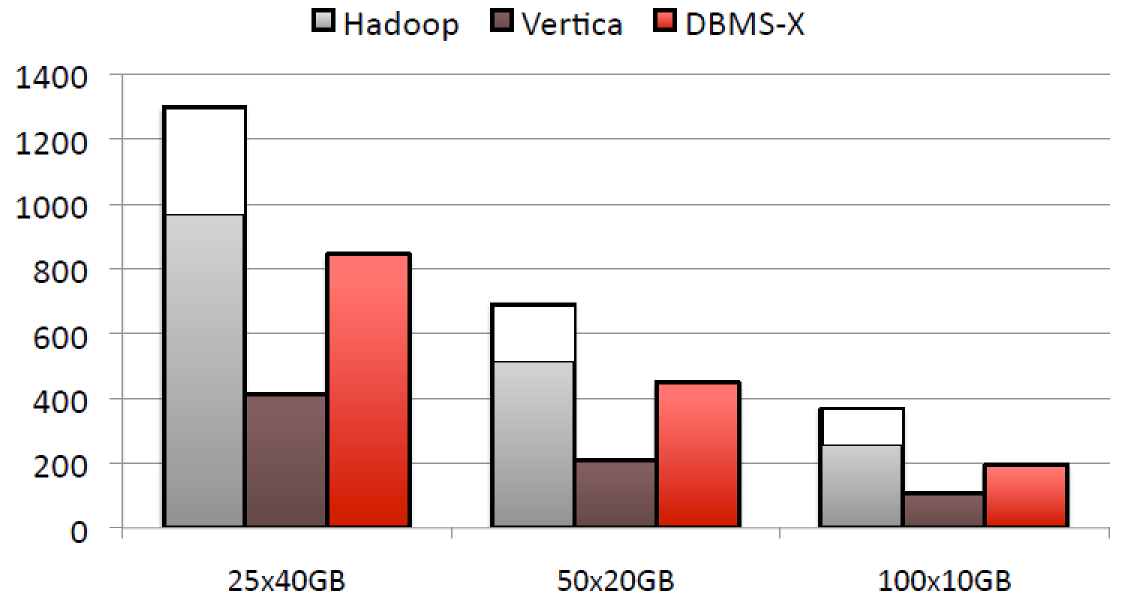
Benchmark 1: Grep Task

- Grep task: Shows overhead of data loading in DBMS
  - Search 3 byte pattern across 10Billion records
  - Each record: 100 bytes (10-byte key, 90-byte value)
  - 1TB across 25,50 or 100 nodes

DBMS slow during data Loading

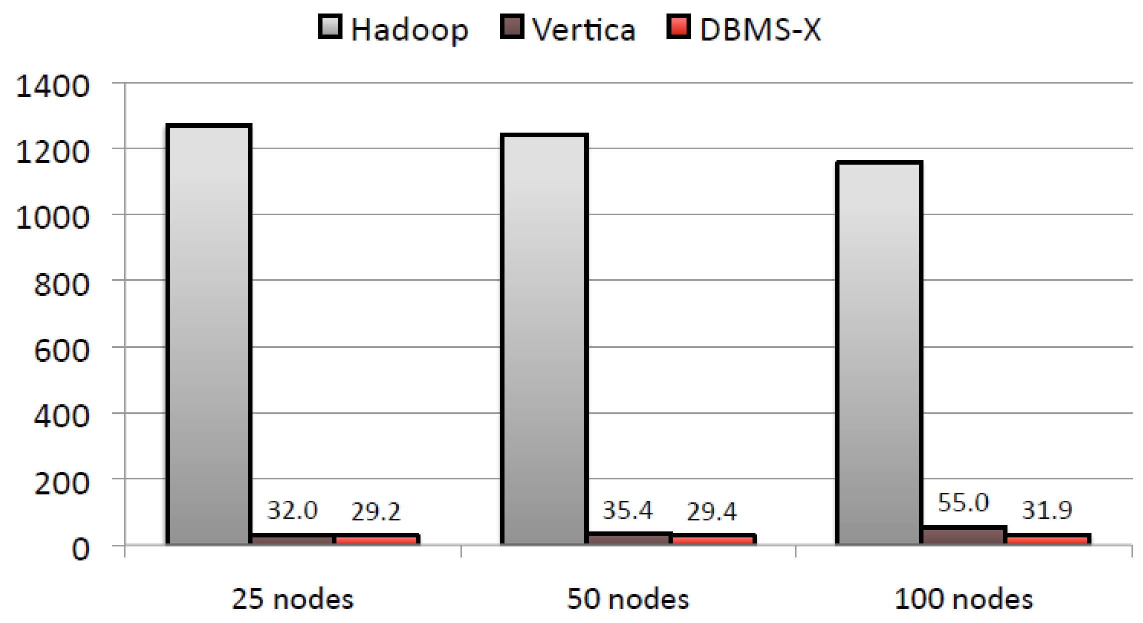


DBMS fast during execution



Analytical Task

- Web processing: Shows the benefit of query optimization
  - 600k html documents
  - 155Million uservisits records, 18 million rankings records
  - Task: Find sourceIP that generated most revenue with avg. pagerank
  - DBMS: Complex SQL join query, MR: 3 separate MR programs



MapReduce: Not a silver bullet

- MapReduce does not fit many cases.
  - Interactive computations
    - E.g. not a user-facing web site back-end.
  - Small updates to big data ***(more on OLTP and this later)***
    - Add a few documents to a big index
  - Small data, since overheads are high.

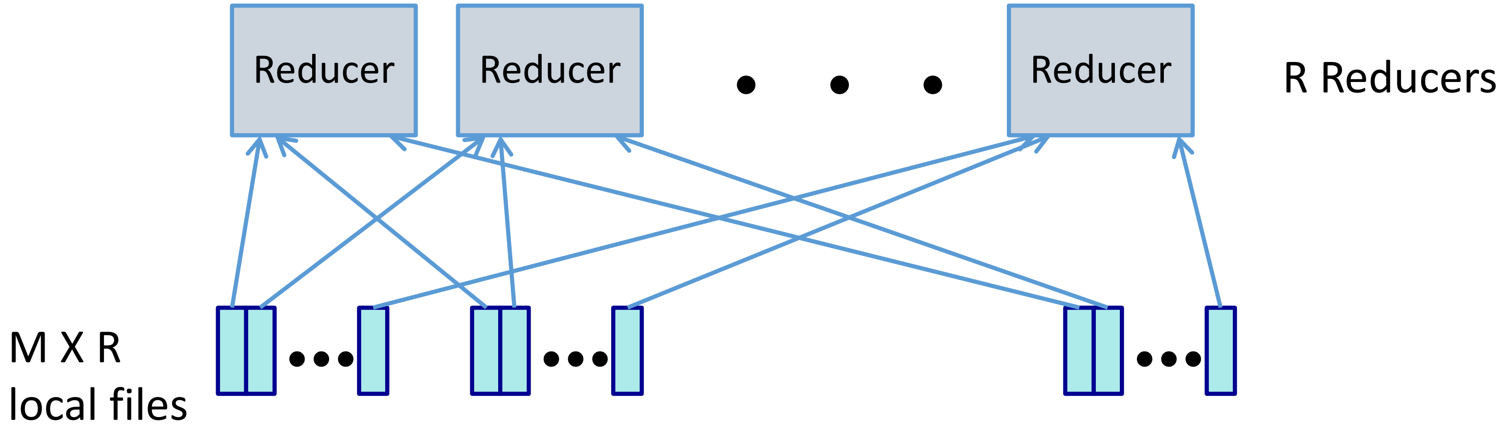
Relational operations over MR: Natural Join

- **Let's look at two relations** R(A, B) **and** S(B, C)
  - We must find tuples that agree on their B components
  - We shall use the B-value of tuples from either relation as the key
  - The value will be the other component and the name of the relation
  - That way the reducer knows each tuple's relation
- **Map:**
  - For each tuple *(a, b)* of R emit the key/value pair *(b, ('R', a))*
  - For each tuple *(b, c)* of S emit the key/value pair *(b, ('S', c))*
- **Reduce:**
  - Each key b will be associated to a list of pairs that are either *('R', a)* or *('S', c)*
  - Emit key/value pairs of the form $(b, [(a_1, b, c_1), (a_2, b, c_2), \quad , (a_n, b, c_n)])$

Shuffle overhead

Each Reducer:

- Handles 1/R of the possible key values
- Fetches its file from each of M mappers => ***Shuffle***
- Sorts all of its entries to group values by keys



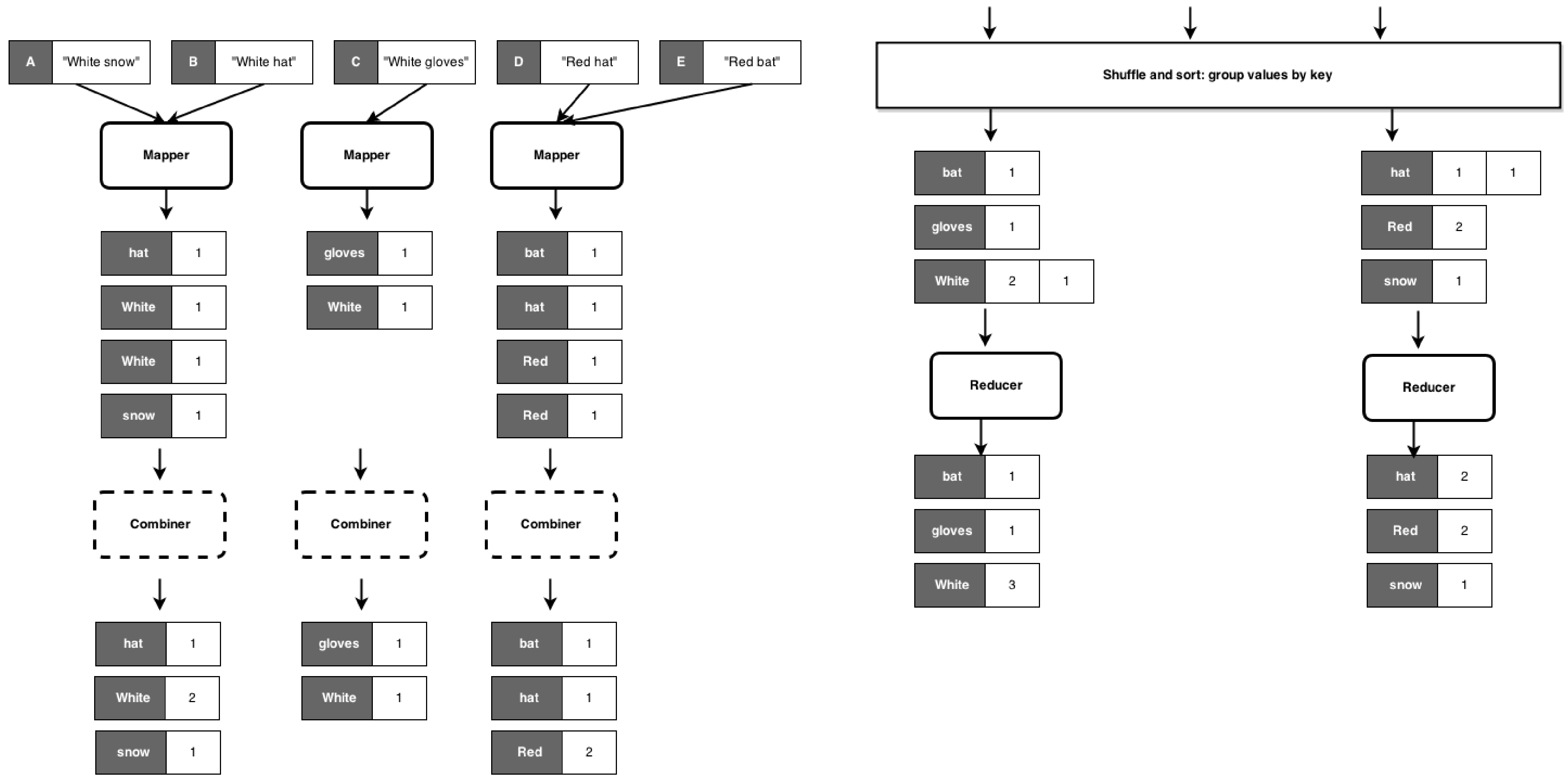
- Shuffle is an all-to-all communication that can overload the network
  - Paper's root switch: 100 to 200 gigabits/second
  - 1800 machines, so 55 megabits/second/machine.
  - Small, e.g. much less than disk (~50-100 MB/s at the time) or RAM speed.

Combiner optimization

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
    - E.g., popular words in Word Count

- Save network time by pre-aggregating at mapper with ***combiner function***
    - Decreases size of intermediate data transferred during shuffle
    - Reduce network load

Combiner example: Word count



Combiner: Algorithmic correctness

- The use of combiners must be thought carefully
  - the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
- Commutative and Associative computations
  - Reducer and Combiner code may be interchangeable
  - This is not true in the general case
- Counter example: Mean
  - We have a large dataset where input keys are strings and input values are integers
    - Dataset can be a log from a website, where the keys are user IDs and values are some measure of activity
  - Compute the mean of all integers associated with the same key

Baseline approach for mean

- We use an **identity mapper**, which groups and sorts appropriately input key-value pairs
  - Reducers keep track of running sum and the number of integers encountered
  - The mean is emitted as the output of the reducer, with the input string as the key

```
class Mapper
    method Map(string $t$, integer $r$)
        Emit(string $t$, integer $r$)

class Reducer
    method Reduce(string $t$, integers $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ do
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, integer $r_{avg}$)
```

Mean with combiners: Caution

- Note: operations are not distributive
  - Mean(1,2,3,4,5) != Mean(Mean(1,2), Mean(3,4,5))
  - Hence: a combiner cannot output partial means and hope that the reducer will compute the correct final mean

- Rule of thumb:
  - Combiners are optimizations, the algorithm should work even when “removing” them

Mean with combiners

```
class MAPPER
    method MAP(string $t$, integer $r$)
        EMIT(string $t$, pair $(r, 1)$)
class COMBINER
    method COMBINE(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        EMIT(string $t$, pair $(sum, cnt)$)
class REDUCER
    method REDUCE(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        EMIT(string $t$, integer $r_{avg}$)
```

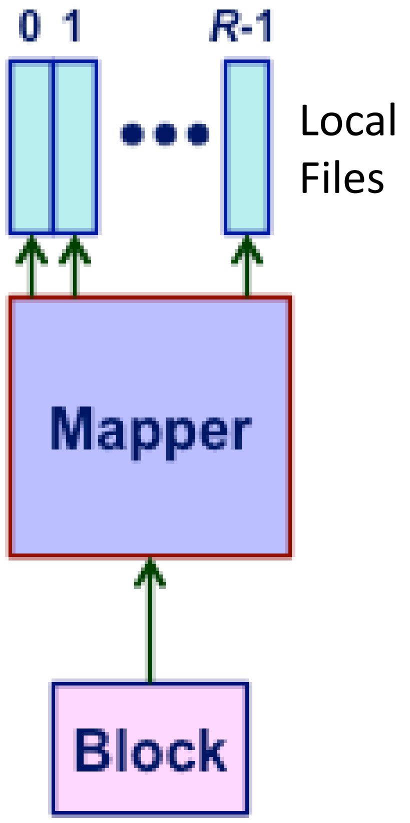
MapReduce: Not a silver bullet

- MapReduce does not fit many cases.
  - Interactive computations
    - E.g. not a user-facing web site back-end.
  - Small updates to big data (more on OLTP and this later)
    - Add a few documents to a big index
  - Small data, since overheads are high.
  - **Unpredictable reads (neither Map nor Reduce can choose input) & load balancing issues**
    - Critical to scaling -- bad for N-1 servers to wait for 1 to finish.
    - Might be bug, flaky hardware, or poor partitioning
    - Leads to ***Stragglers:*** Tasks that take long time to execute

Better load balancing (1): Custom partitioning

$$\begin{tikzcd} K \arrow[r] & \boxed{h} \arrow[r] & h(K) \in \{0,\dots,R-1\} \end{tikzcd}$$

- Mapper Operation
  - Reads input file blocks
  - Generates pairs ⟨*K*, *V*⟩
  - Writes to local file *h*(*K*)
- Hash Function *h* partitions intermediate key space K
  - Default *h*: Maps each key *K* to integer *i* such that $0 \le i < R$
- Can also specify a customized partitioning function
  - Ex: output keys are URLs, we want all entries for a single host to end up in the same output file.
  - Can use "*hash(Hostname(urlkey)) mod R*" as *h*



Better load balancing (2): Task duplication

- Assign many more tasks than workers.
  - Master hands out new tasks to workers who finish previous tasks.
  - So no task is so big it dominates completion time (hopefully).
  - So faster servers do more work than slower ones, finish about the same time.

MapReduce: Not a silver bullet

- MapReduce does not fit many cases.
  - Interactive computations
    - E.g. not a user-facing web site back-end.
  - Small updates to big data (more on OLTP and this later)
    - Add a few documents to a big index
  - Small data, since overheads are high.
  - Unpredictable reads (neither Map nor Reduce can choose input) & load balancing issues
    - Critical to scaling -- bad for N-1 servers to wait for 1 to finish.
    - Might be bug, flaky hardware, or poor partitioning
    - Leads to ***Stragglers:*** Tasks that take long time to execute
  - **Iterative computations *(more on this in the next lecture)***
    - **Algorithms with multiple rounds, e.g. k-means, page rank**

MapReduce: Not a silver bullet

- MapReduce does not fit many cases.
  - Interactive computations
    - E.g. not a user-facing web site back-end.
  - Small updates to big data (more on OLTP and this later)
    - Add a few documents to a big index
  - Small data, since overheads are high.
  - Unpredictable reads (neither Map nor Reduce can choose input) & load balancing issues
    - Critical to scaling -- bad for N-1 servers to wait for 1 to finish.
    - Might be bug, flaky hardware, or poor partitioning
    - Leads to ***Stragglers:*** Tasks that take long time to execute
  - Iterative computations *(more on this in the next lecture)*
    - Algorithms with multiple rounds, e.g. k-means, page rank
- **“MapReduce: A major step backwards” – Dewitt, Stonebraker**

RDBMS vs MapReduce: Summary

- Systems designed to meet different requirements
- Traditional relational databases
  - Interactive SQL analytics
    - Optimized for point queries (random access) and range queries (scans)
  - Built for enterprises (dedicated DB admin, few DB servers)
    - No need to scale to 1,000 or more nodes
    - Proprietary and paid products
  - Fine-grained updates to shared data
    - Guaranteeing ACID properties despite concurrent access and failures
- MapReduce
  - Latency-insensitive batch analytics
    - Sequential scans of Petabytes of data
  - Built for the cloud: Fault tolerance across commodity servers
    - Focus on faults during query rather than recovery after updates
  - Open source and “One person” deployment
    - Turn any Java developer into a distributed analytics engineer