COMP 430
Intro. to Database Systems

Query optimization

Slides use ideas from Amol Deshpande, Chris Ré.

Questions we want to address

- How to understand queries' performance?
- What queries can system optimize for us?
  - How can we tell?
  - ~~How does it optimize?~~
- What queries do we need to optimize ourselves?
  - How can we tell?
  - How do we optimize?

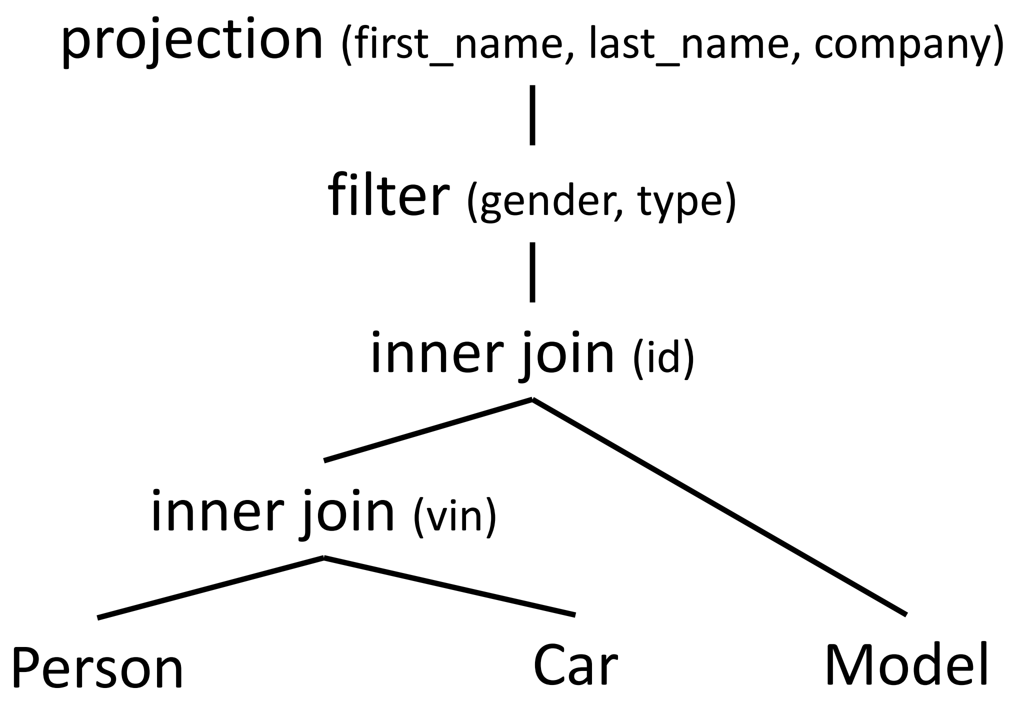
Need to understand some basic optimization strategies, but not the details.

Example query

```
SELECT Person.first_name, Person.last_name, Model.company
FROM Person
INNER JOIN Car ON Person.car_vin = Car.vin
INNER JOIN Company ON Car.model_id = Model.id
WHERE Person.gender = ‘F’ AND Car.type IN (‘sedan’, ‘coupe’);
```

A simple query tree

```sql
SELECT Person.first_name, Person.last_name, Model.company
FROM Person
INNER JOIN Car ON Person.car_vin = Car.vin
INNER JOIN Company ON Car.model_id = Model.id
WHERE Person.gender = ‘F’ AND Car.type IN (‘sedan’, ‘coupe’);
```

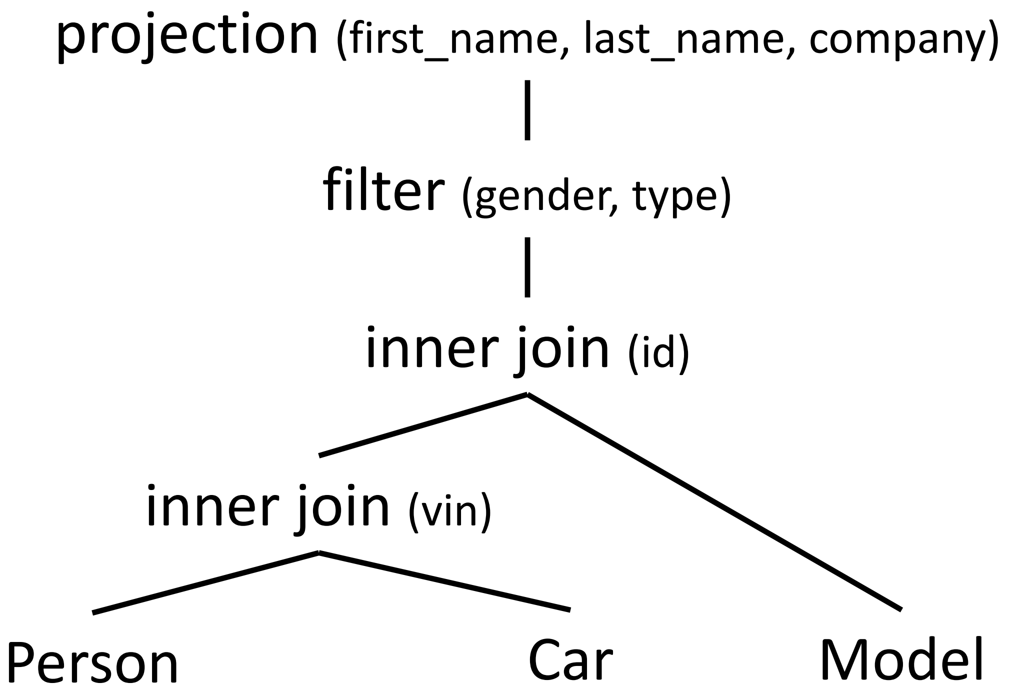


Tree form of *relational algebra* expression

Activity – Generate equivalent query trees

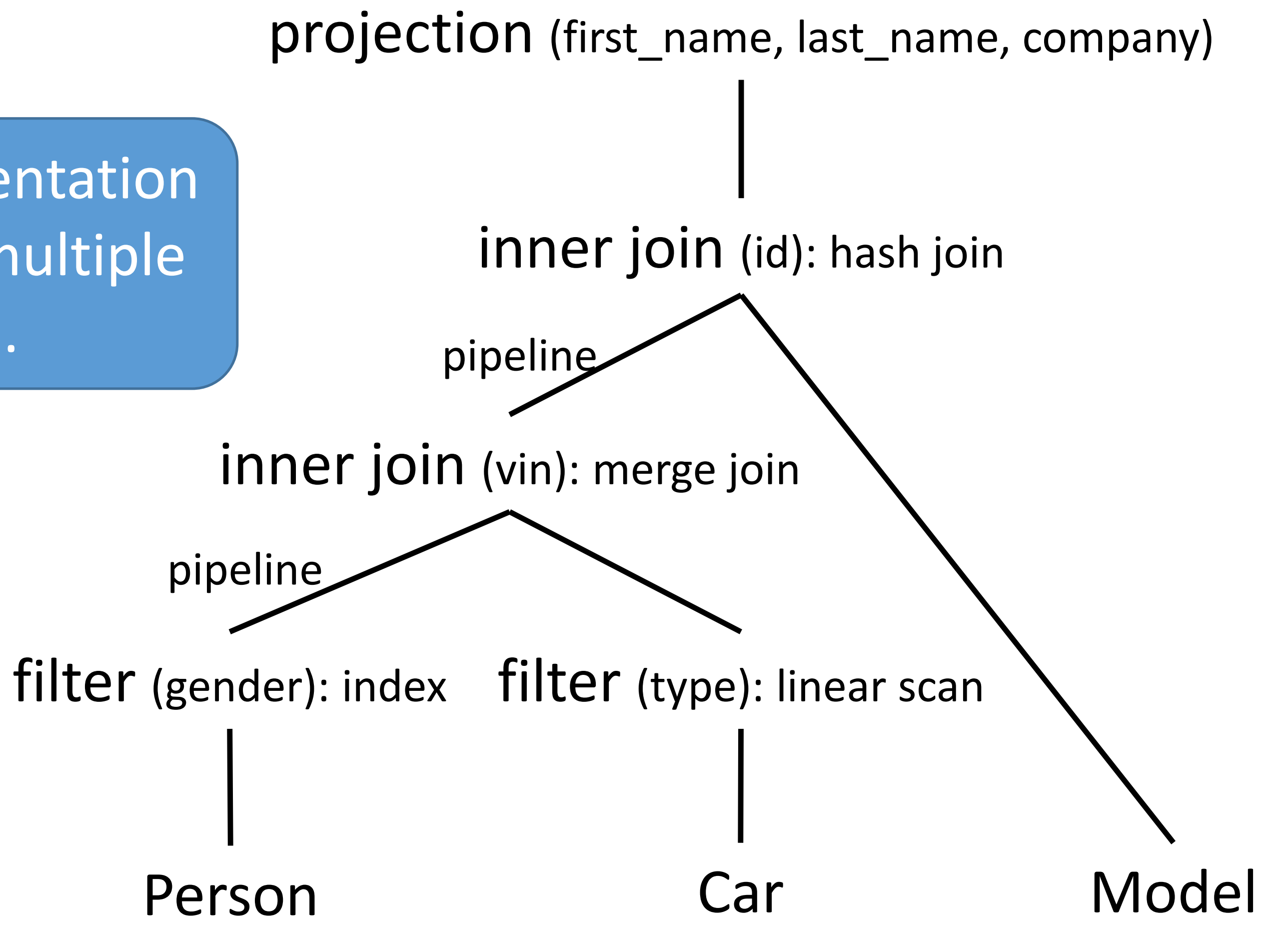
```
SELECT Person.first_name, Person.last_name, Model.company
FROM Person
INNER JOIN Car ON Person.car_vin = Car.vin
INNER JOIN Company ON Car.model_id = Model.id
WHERE Person.gender = ‘F’ AND Car.type IN (‘sedan’, ‘coupe’);
```

Suggestions?



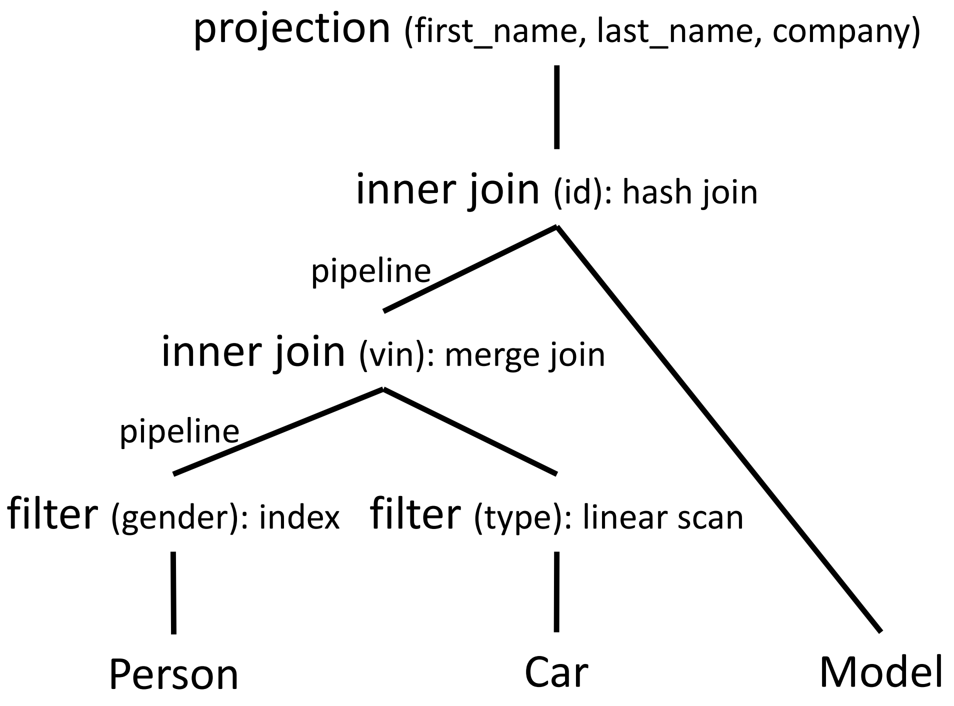
Generate execution plans

Notate with implementation methods. Possibly multiple plans per tree.



Analyze costs of each plan

- Algorithm cost for each node
- Data amount
  - #tuples
  - Total #bytes or #disk blocks
  - Selectivity of filters
- Data access
  - Method – index, linear scan, binary search
  - Source – memory, disk



Data-dependent – determined at run time.
Use & cache lowest cost plan.

Trees → Plans → Costs

Steps integrated

- Can be too many trees/plans to generate all of them.
- Cost heuristics guide tree/plan generation.

Simple example of cost analysis

```
SELECT *
FROM Person
WHERE name IN (‘Greiner’, ‘Halverhout’) AND
      zipcode = 77055 AND
      birthdate > ‘19670212’;
```

Test	CPU cost/record
name IN	100ns
zipcode =	1ns
birthdate >	1000ns

400,000,000 records

Simple example of cost analysis: Test order

For each Person record:
If name IN …
Then if zipcode = …
Then if birthdate > …
Then output record

How many possible orders?
How to calculate cost for each?

Simple example of cost analysis: Test order

400,000,000 records

Test	CPU cost/record	Selectivity	Cost for this test	…simplified
name IN	100ns	0.00001	400,000,000 × 100ns	40,000,000,000ns
zipcode =	1ns	0.0001	400,000,000 × 0.00001 × 1ns	4,000ns
birthdate >	1000ns	0.3	400,000,000 × 0.00001 × 0.0001 × 1000ns	400ns
				40,000,004,400ns

Test	CPU cost/record	Selectivity	Cost for this test	…simplified
zipcode =	1ns	0.0001	400,000,000 × 1ns	400,000,000ns
name IN	100ns	0.00001	400,000,000 × 0.0001 × 100ns	400,000ns
birthdate >	1000ns	0.3	400,000,000 × 0.00001 × 0.0001 × 1000ns	400ns
				400,400,400ns

Simple example of cost analysis: Test order

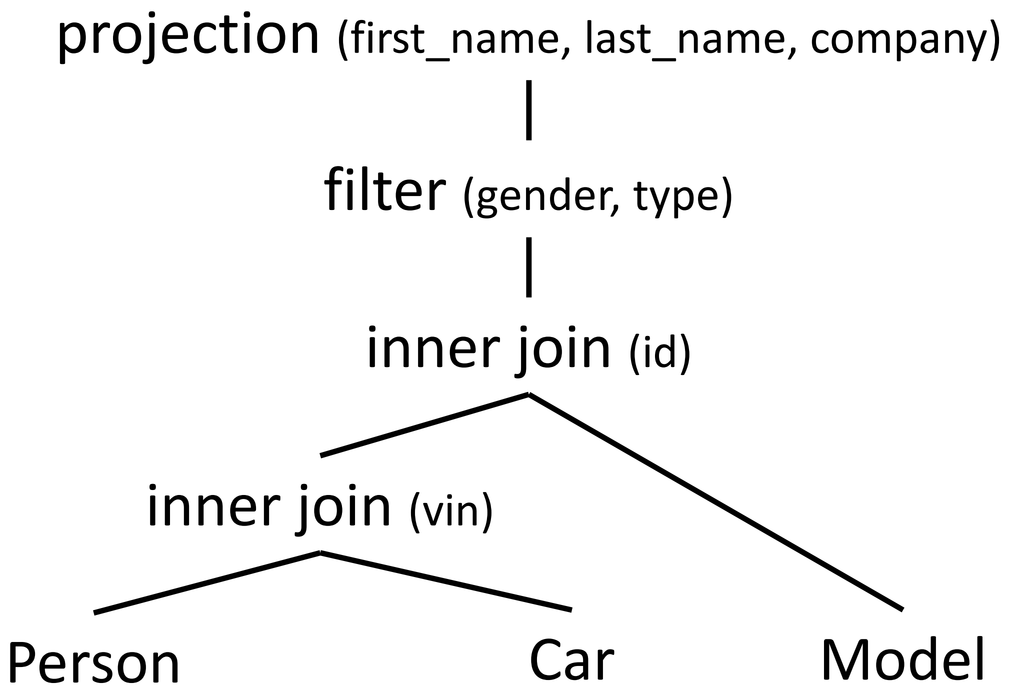
How to search through all the plans efficiently?

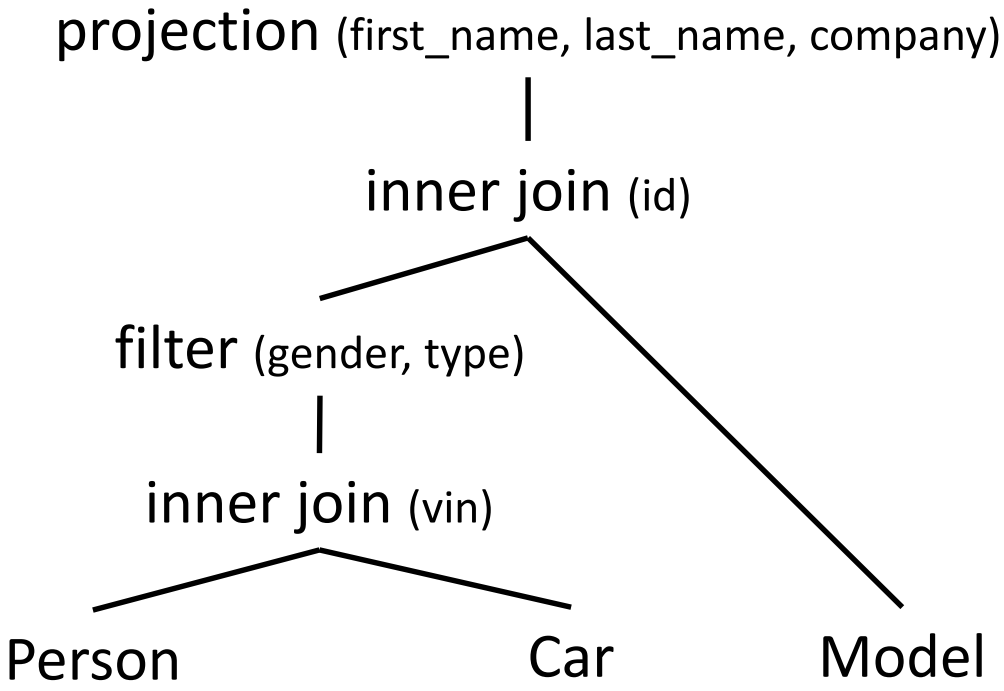
Best: in decreasing order by $\frac{1 - \text{Selectivity}(\text{predicate})}{\text{Cost}(\text{predicate})}$,

i.e., the fraction of data eliminated per unit time.

Transforming query trees

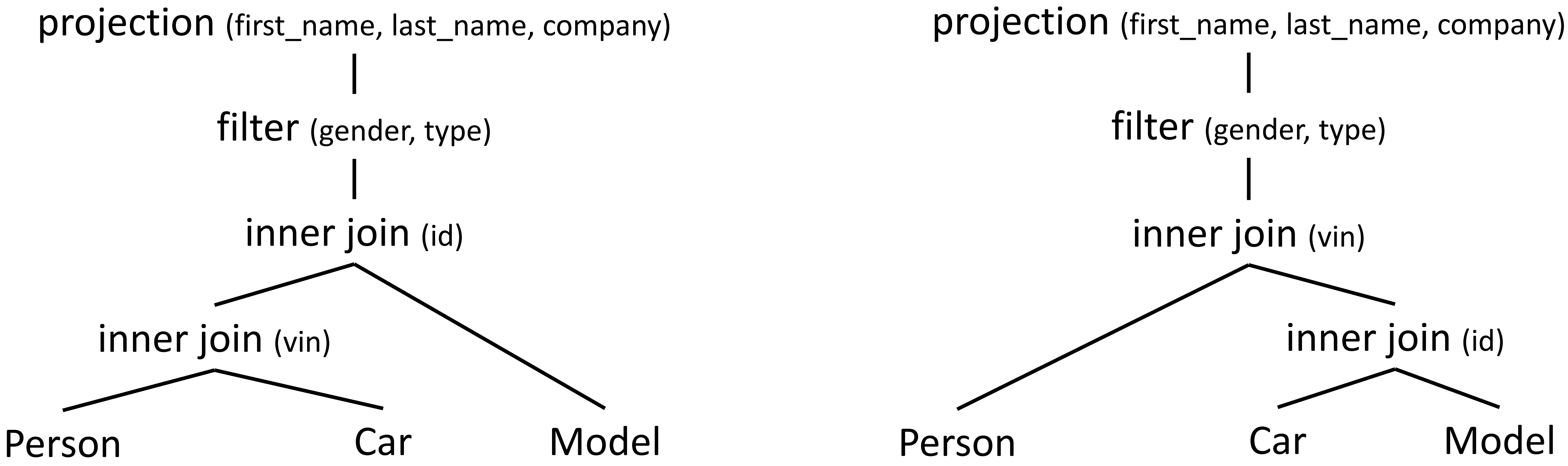
Relational algebra equivalences: example





$\text{filter}(\text{inner join}(A,B)) = \text{filter}_{AB}(\text{inner join}(\text{filter}_A(A),\text{filter}_B(B))$

Relational algebra equivalences: example



Inner joins are associative & commutative.

Too many trees to enumerate all

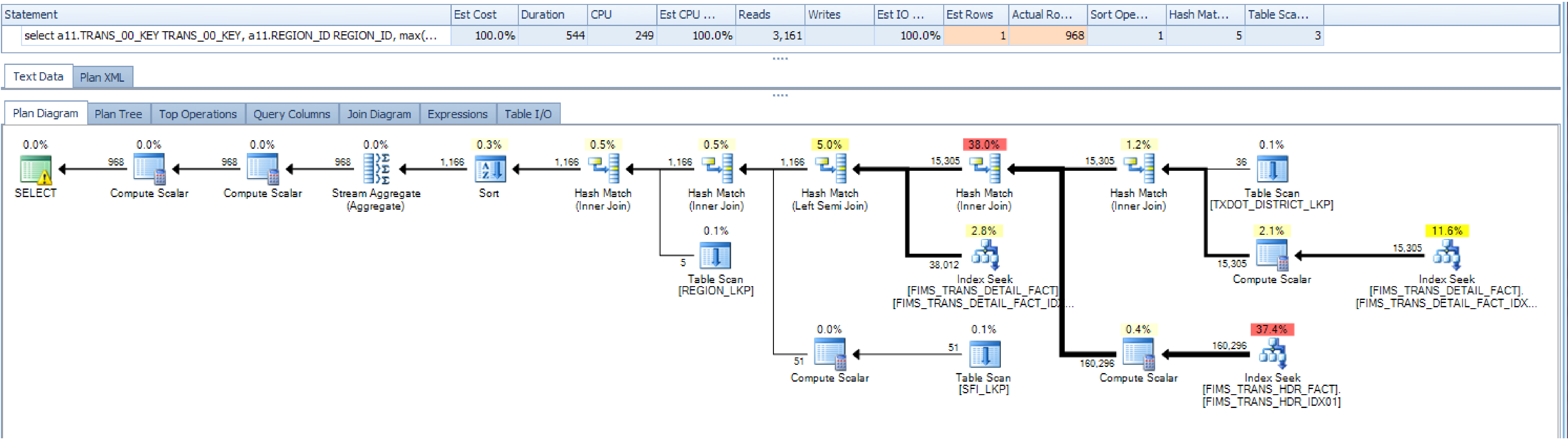
Example: inner join of n tables with commutativity & associativity $\rightarrow$ how many trees?

$$n!\ 2^n$$

Implementation Strategies

Viewing & understanding execution plan

Helpful to be able to understand plan and any bottlenecks.



Scan implementations

- Indexing – previously discussed
- Binary search – if sorted on search key
- Linear search

What to do with each operation's results?

- *Materialization*
  - Data saved to disk (or memory, if small enough)

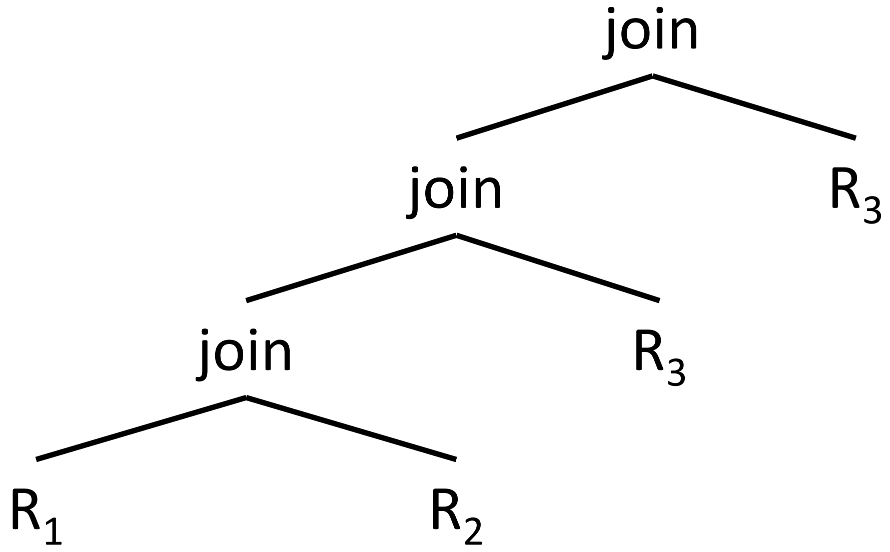
  \- Disk access
  \+ Can reuse data if needed multiple times

- Pipelining
  - Consumer uses data as producer generates it
  - Use buffer to allow consumer to be faster than producer

  \+ No disk access

Left-deep trees have one long pipeline

Common to consider only *left-deep* trees of joins.



Join implementations

Join(R, S) on R.A = S.A

Let T(R) = #tuples in R, B(R) = #blocks in R

- Three main strategies
  - Nested joins + 2 variants
  - Sort-merge joins + 1 variant
  - Hash joins

Nested loop join

```
For r in R:
    For s in S:
        If r.a == s.a, yield (r,s)
```

Cost $= B(R) + T(R)B(S) + B(\text{result})$

```
For s in S:
    For r in R:
        If r.a == s.a, yield (r,s)
```

Cost $= B(S) + T(S)B(R) + B(\text{result})$

Block nested loop join – I/O aware

Let B+1 = #blocks in memory

```
For each group of B-1 blocks br of R:
    For block bs of S:
        For r in br:
            For s in bs:
                If r.a == s.a, yield (r,s)
```

$$\text{Cost} = B(R) + \frac{B(R)}{B-1} B(S) + B(\text{result})$$

Choose R = smaller relation

Index nested loop join

Assumes S indexed on A.

```
For r in R:
   If r.a == index(s,a), yield (r,s)
```

$$\text{Cost } = B(R) + T(R)C + B(\text{result})$$

C = I/O to access all distinct values in index.
Not constant, but typically < 10.

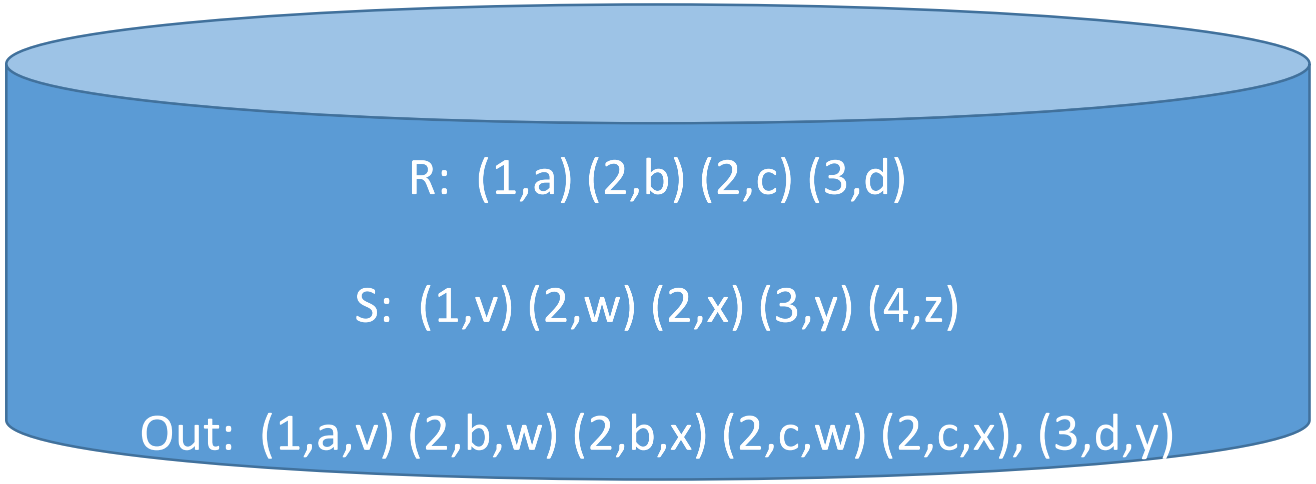
Sort-merge join

1. Sort R and S each on A.
   - Use *external merge sort* (disk-based).
   - But, R and/or S might already be sorted on A.
2. "Merge" R and S.
   - Linear scan R.
   - Linear scan S, backing up on duplicates.

$$R\left(\log_B \frac{R}{B+1}\right)$$

$$S\left(\log_B \frac{S}{B+1}\right)$$

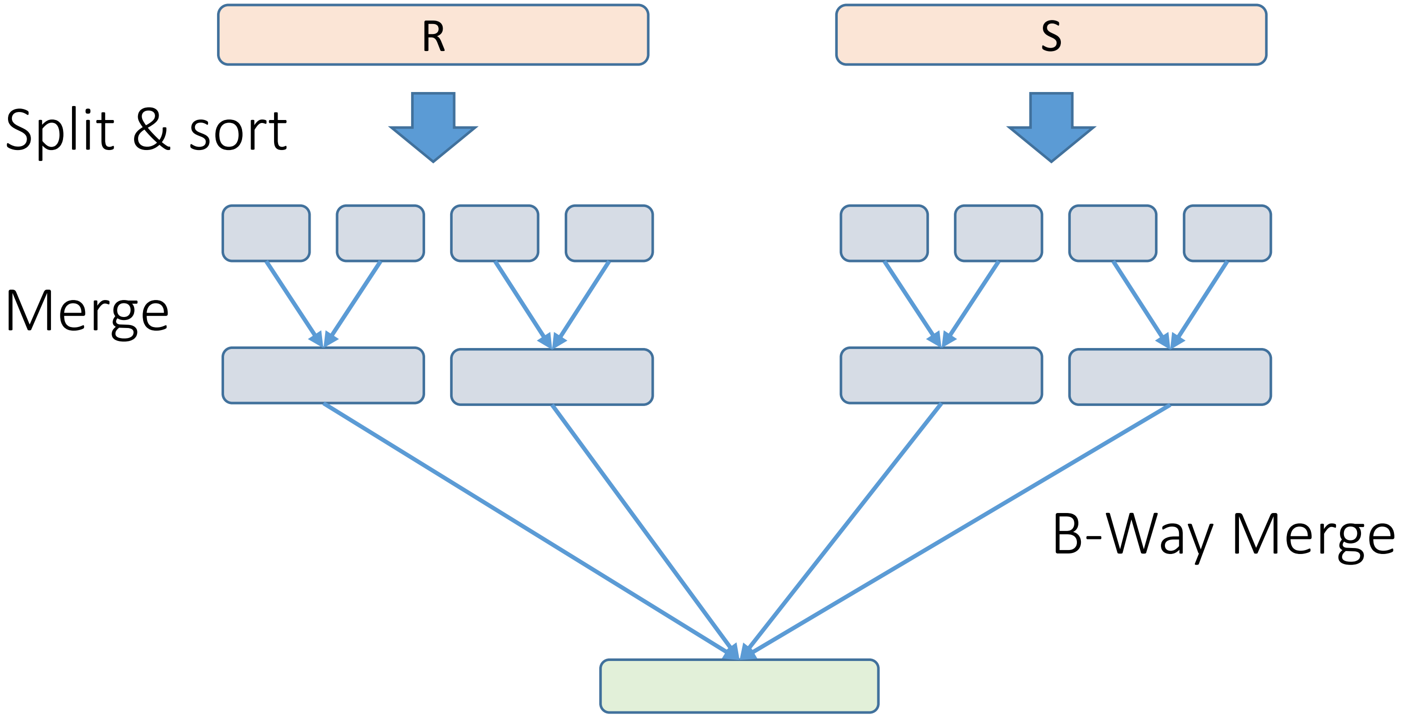
$$B(R) + B(S) \quad \dots \quad B(R)B(S)$$



Not too sensitive to skew.
Leaves data sorted!

Sort-merge join – I/O aware

Integrate the mergesort's merging with the "merge" of R,S

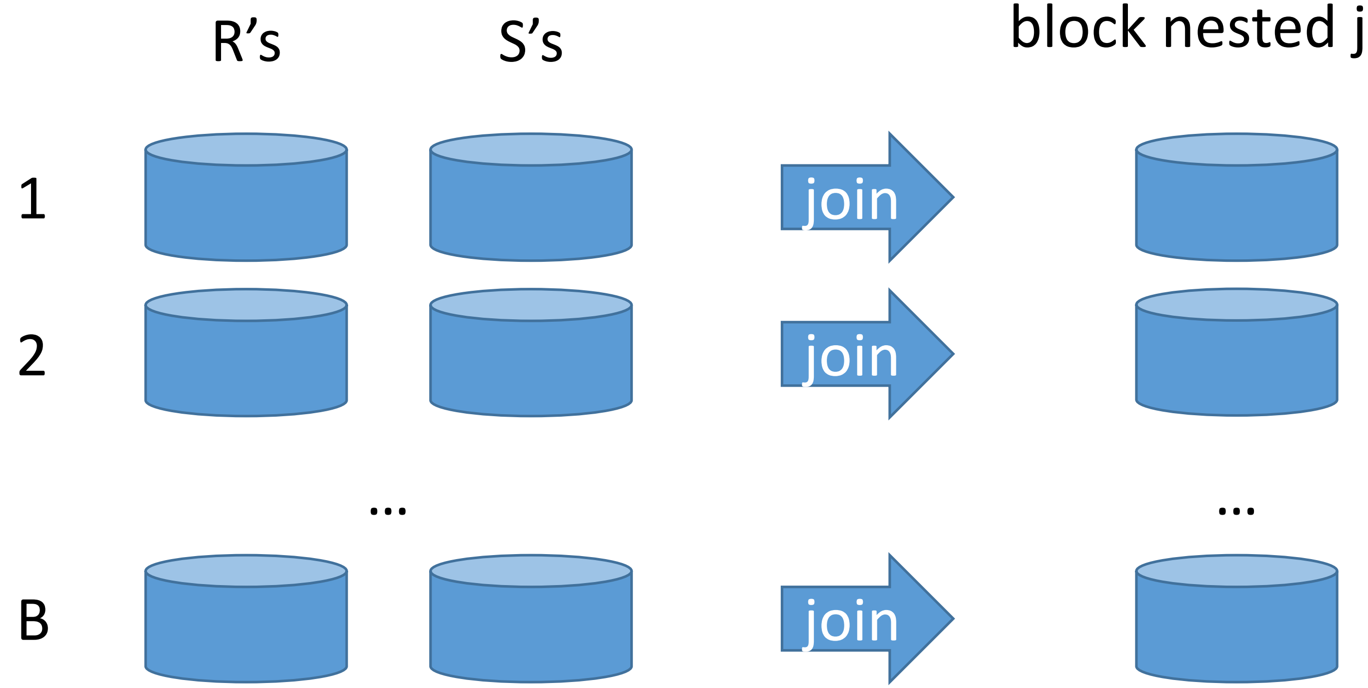


Hash-join

Use hashing to decompose one large join into many small joins.

Step 1: Hash R, S each on same hash function into B buckets.

Step 2: Join pairs of buckets. Use hash-join recursively or block nested join.



Highly parallelizable

Size estimation

Costs depend on amount of data

Estimating sizes

- Size of each relation
- Selectivity of each join & filter condition

- How calculated?
  - Gather statistics
  - Use statistics to estimate selectivity

Size statistics per relation R

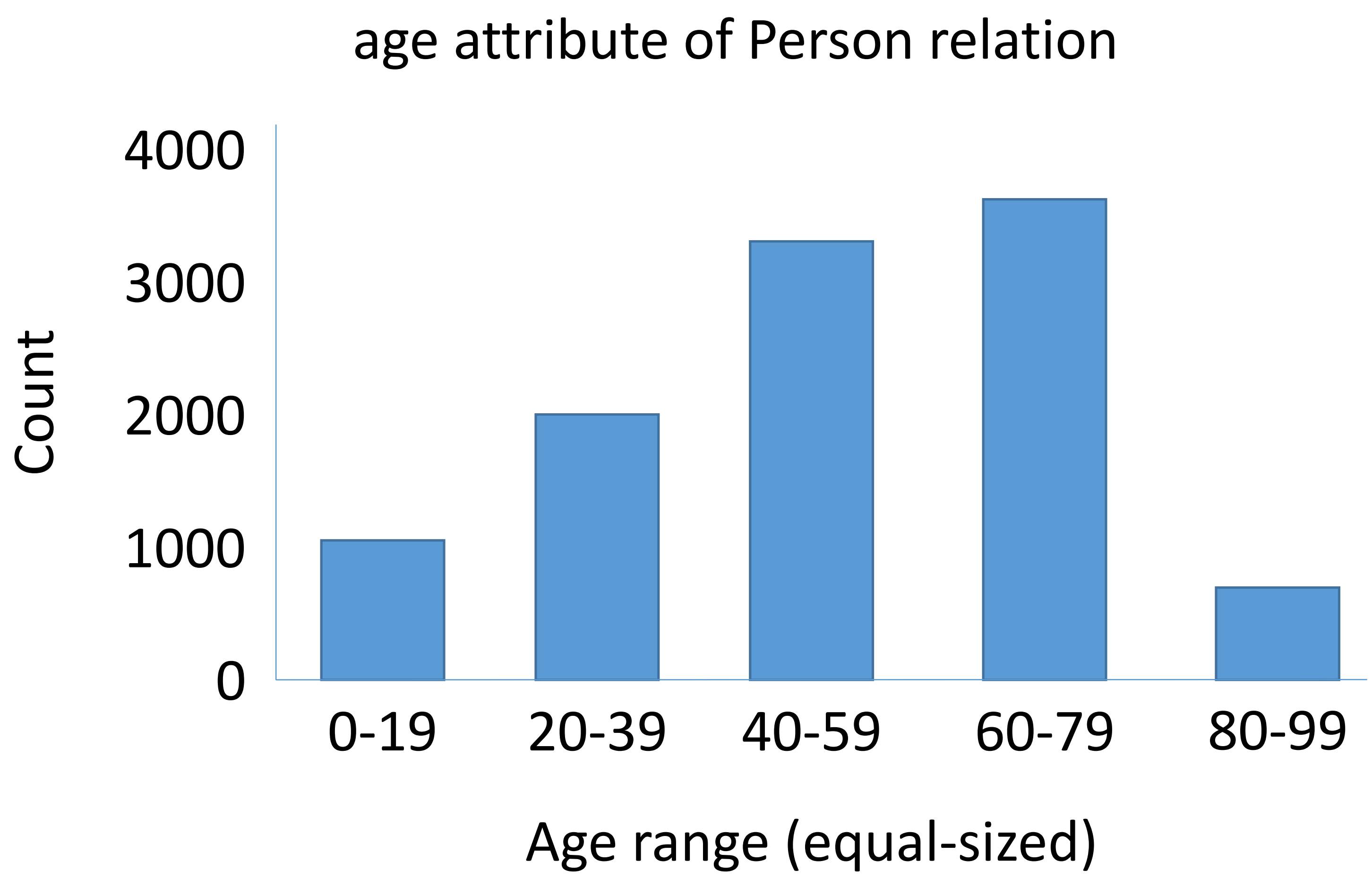
- $tuples(R)$: #tuples
- $length(R)$: #bytes per tuple
- $bfactor(R)$: #tuples that fit into a block (blocking factor)
- $blocks(R)$: #blocks – if no fragmentation, $= \left\lceil \frac{tuples(R)}{bfactor(R)} \right\rceil$

Value statistics for attribute A

- $values(A)$: #distinct values
- $minval(A)$: minimum value
- $maxval(A)$: maximum value

- Distribution of values for attribute A – estimated by histogram

Equi-width histogram example



Histograms summary

- Various other types of histograms

- High-level goals
  - Time- & Space-efficient to compute
  - Useful in estimating selectivity of equality and range conditions

Managing statistics

- Accuracy vs. resources
  - Update frequency
  - Build histograms for which attributes, with how much accuracy

- Automatically by DBMS
- Some manual control
  - SQL Server: CREATE STATISTICS, UPDATE STATISTICS

- Can view statistics
  - SQL Server: DBCC SHOW_STATISTICS

Size estimates for various operations

- CrossJoin(R,S) $tuples(R) \times tuples(S)$
- InnerJoin(R,S) $tuples(R)$
- OuterJoin(R,S) $tuples(R) + tuples(S)$

Assuming join on R's FK & S's PK

- R∪S $tuples(R) + tuples(S)$
- R∩S $\min\big(tuples(R), tuples(S)\big)$
- R-S $tuples(R)$

Calculating selectivity on filter conditions

- WHERE R.A = v:
  - Without histogram: $s = \frac{1}{values(R.A)}$
  - With equal-width histogram: $s = \frac{count(range\_containing\_A, R)}{tuples(R) \times range\_width}$

Calculating selectivity on filter conditions

- WHERE R.A = S.B
  - Without histograms: $s = \min\left(\frac{1}{values(R.A)}, \frac{1}{values(S.B)}\right) = \frac{1}{\max(values(R.A), values(S.B))}$
  - With histograms: Generalize the previous.

R.A values(R.A)=3	S.B values(S.B)=2	Match?
1	1	Yes
2	2	Yes
3	1	No
1	2	No
2	1	No
3	2	No

I.e., expect 1/3 to match.

Calculating selectivity on filter conditions

- WHERE R.A $\geq$ v:
  - Without statistics: $s = \frac{1}{2}$
  - Without histogram: $s = \begin{cases} 0 & \text{if } v > maxval(R.A) \\ \frac{maxval(R.A) - v}{maxval(R.A) - minval(R.A)} & \text{otherwise} \end{cases}$
  - With equal-width histogram: Generalize the previous.

Calculating selectivity on filter conditions

- Conjunction: $s = \prod s_i$
- Disjunction: $s = 1 - \prod (1 - s_i)$
- Negation: $s = 1 - s'$

User-level Query Optimization

So, what should **you** the SQL programmer do?

Just a bit of terminology

- Query optimization – what the system does
- Query tuning – what the programmer does

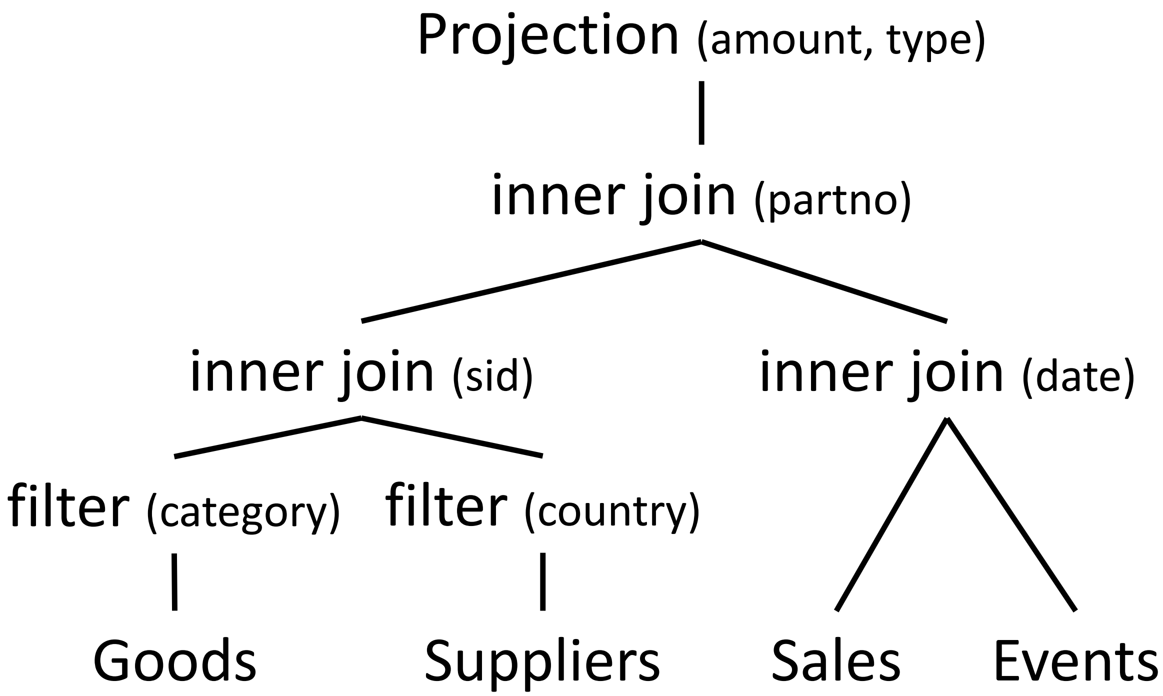
Understanding current performance

What is the bottleneck? Why?

- Query plan
- Statistics – DBCC SHOW_STATISTICS
- Estimate statistics yourself
- Experiment with timings

How would you investigate slow query?

```
SELECT Sales.amount, Events.type
FROM Sales, Events, Goods, Suppliers
WHERE Sales.date = Events.date AND
          Sales.partno = Goods.partno AND
          Suppliers.sid = Goods.sid AND
          Goods.category = 'engine' AND
          Suppliers.country = 'US'
```



Technique summary

- Improve system resources
- Improve query optimization

- Don’t get in optimizer’s way
- **Reduce computation**
- **Reduce data usage**

- Denormalization – next topic
- Eliminating constraints & triggers – generally disrecommended

Not a prioritized list!

Improving system resources

Not always feasible, but can improve EVERY query.

- Faster CPU
- More physical memory
  - Reduce need for I/O
- Faster I/O
  - SSDs instead of disks
  - Distribute over multiple drives to increase bandwidth

Improving query optimization

Help system optimize better.

- Use simpler synthetic keys
- Create indices – CREATE INDEX
- Manage index growth – FILL FACTOR
- Create specialized statistics – CREATE/UPDATE STATISTICS
- Partition or data stripe table to improve I/O

- Check whether benefits outweigh overheads.

Overriding the query optimizer

Sometimes, the query optimizer gets it wrong.

- Force it to use index: SELECT … FROM … WITH INDEX …
- Force join order

However, the query optimizer is usually smarter than you.

Don’t get in optimizer’s way

Use WHERE expressions on the raw attribute

```
SELECT …
FROM Account
WHERE YEAR(created_on) = 2016 AND
          MONTH(created_on) = 1;
```

Blocks usage of both index & statistics.

```
SELECT …
FROM Account
WHERE created_on BETWEEN '1/1/2016' AND '2/1/2016';
```

Other functions commonly used like this: ISNULL() and implicit type conversions.

Don’t loop over SQL calls

```
for each person in person_list:
   INSERT INTO Person VALUES (this person’s data)
```

Use one INSERT INTO with a list of values.

Such examples generally stem from not understanding SQL’s capabilities.

```
product_list = SELECT * FROM Product

for each product in product_list:
   number_product_sold = SELECT COUNT(product_id)
                         FROM Sales
                         WHERE product_id = this product’s id
```

Use one SELECT with a join, grouping on product_id.

Don’t use cursors unnecessarily

Many newbie examples just duplicate SELECT’s features.

Reduce computation

Avoid constant-valued subqueries

```
SELECT first_name, last_name
FROM Person
WHERE age = (SELECT Max(age)
             FROM Person);
```

→

```
SELECT @oldest_age = MAX(age)
FROM Person;

SELECT first_name, last_name
FROM Person
WHERE age = @oldest_age;
```

Avoid *correlated* subqueries when possible

```
SELECT name,
       city,
       (SELECT company_name FROM Company WHERE id = Customer.company_id)
FROM Customer;
```

```
SELECT name,
       city,
       company_name
FROM Customer
INNER JOIN Company ON Company.id = Customer.company_id;
```

Avoid *correlated* subqueries when possible

```
SELECT id, first_name, last_name, salary
FROM Employee e
WHERE EXISTS (SELECT 1
              FROM Orders o
              WHERE e.id = o.sales_rep_id AND
                    o.customer_id = 123);
```

```
SELECT id, first_name, last_name, salary
FROM Employee e
WHERE id IN (SELECT sales_rep_id
             FROM Orders
             WHERE o.customer_id = 123);
```

Avoid DISTINCT when it's unnecessary

Find departments with employees

```
SELECT DISTINCT Dept.id, Dept.name
FROM Dept, Employee
WHERE Dept.id = Employee.dept_id;
```

```
SELECT id, name
FROM Dept
WHERE id IN (SELECT dept_id
                 FROM Employee);
```

A poor variation:

```
SELECT id, name
FROM Dept
WHERE id IN (SELECT DISTINCT dept_id
                 FROM Employee);
```

Similarly, avoid UNION when UNION ALL is sufficient.

Don’t COUNT to check for existence

```
SELECT id, name
FROM Dept
WHERE (SELECT COUNT(*)
           FROM Employee
           WHERE Dept.id = Employee.dept_id)
          > 0;
```

```
SELECT id, name
FROM Dept
WHERE id IN (SELECT dept_id
                     FROM Employee);
```

Use CASE to avoid multiple passes

```
SELECT COUNT (*)
FROM Employee
WHERE salary < 20000;

SELECT COUNT (*)
FROM Employee
WHERE salary BETWEEN 20000
                     AND 50000;

SELECT COUNT (*)
FROM Employee
WHERE salary > 50000;
```

```
SELECT COUNT (CASE WHEN salary < 20000
                              THEN 1 ELSE null END),
          COUNT (CASE WHEN salary BETWEEN 2000
                                              AND 50000
                              THEN 1 ELSE null END),
          COUNT (CASE WHEN salary > 50000
                              THEN 1 ELSE null END);
```

Don’t ORDER BY unnecessarily

- Don’t sort in nested/helper queries. Only sort your final results.
- Watch for this when using a pre-existing query as a helper.

Reducing data usage

Don’t select too much data

```
SELECT first_name, last_name, city
FROM Person
WHERE city = ‘Houston’;
```

```
SELECT first_name, last_name, city
FROM Person;
```

… and then application code filters the data.

```
SELECT *
FROM Person
WHERE city = ‘Houston’;
```

SELECT less data in subquery

```
SELECT id, first_name, last_name, salary
FROM Employee e
WHERE dept_id = 456 AND
        id IN (SELECT sales_rep_id FROM Orders);
```

Probably sorts and/or indexes subquery.

In this case, a correlated subquery can be better.

- Subquery results in less data.
- Subquery can use indices on both attributes.

```
SELECT id, first_name, last_name, salary
FROM Employee e
WHERE dept_id = 456 AND
        EXISTS (SELECT 1
                FROM Orders
                WHERE e.id = o.sales_rep_id);
```

Don’t update data with identical data

```
UPDATE Person
SET active = False
WHERE … AND      /* no recent transactions, website visits, or customer calls */
          active = True;
```

Avoids many writes to Person. Reduces any logging of writes.