

# COMP2400

## Relational Databases

### Lecture 30: Query Processing

---

Ben Lippmeier  
Australian National University  
Semester 2  
2008

# Movie database example

---

movie	
code	: varchar(16)
title	: varchar(80)
year	: int
director	: varchar(80)
producer	: varchar(80)
studio	: varchar(40)
process	: varchar(40)
category	: varchar(40)
awards	: varchar(80)
locale	: varchar(20)
notes	: text

**11470 rows**

casts	
code	: varchar(16)
actor_name	: varchar(40)
actor_role	: text

**44499 rows**

actor	
stage_name	: varchar(80)
given_name	: varchar(40)
last_name	: varchar(40)
sex	: char(1)
birth_year	: int
death_year	: int
role	: varchar(80)
origin	: varchar(40)
notes	: text

**6805 rows**

- Derived from data by Gio Wiederhold, Stanford University.
- Source: UCI KKD Archive. <http://kkd.ics.uci.edu>

# Featuring Clint Eastwood

---

```
SELECT title, year
FROM    movie, casts
WHERE   movie.code = casts.code
          AND    actor_name = 'Clint Eastwood';
```

$\Pi_{\text{title, year}}$

$(\sigma_{\text{actor\_name}='Clint Eastwood'}$

$(\text{movie} \bowtie_{\text{movie.code} = \text{casts.code}} \text{casts}))$

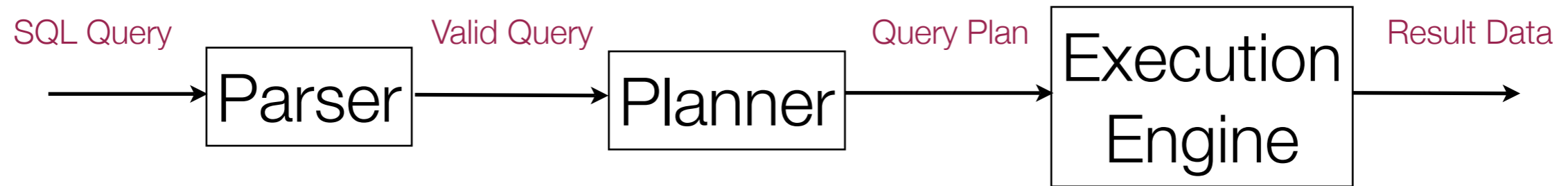
# SQL and Relational Algebra are logical languages.

---

- SQL and Relational Algebra *describe* the desired result of a query.
- They do not instruct the DBMS how to actually compute the result.
- The DBMS must examine the query and decide on a good way of physically reading data from disk and processing it.
- There are often *many* possible ways of satisfying a query.
- Some are orders of magnitude faster than others.

# Query processing

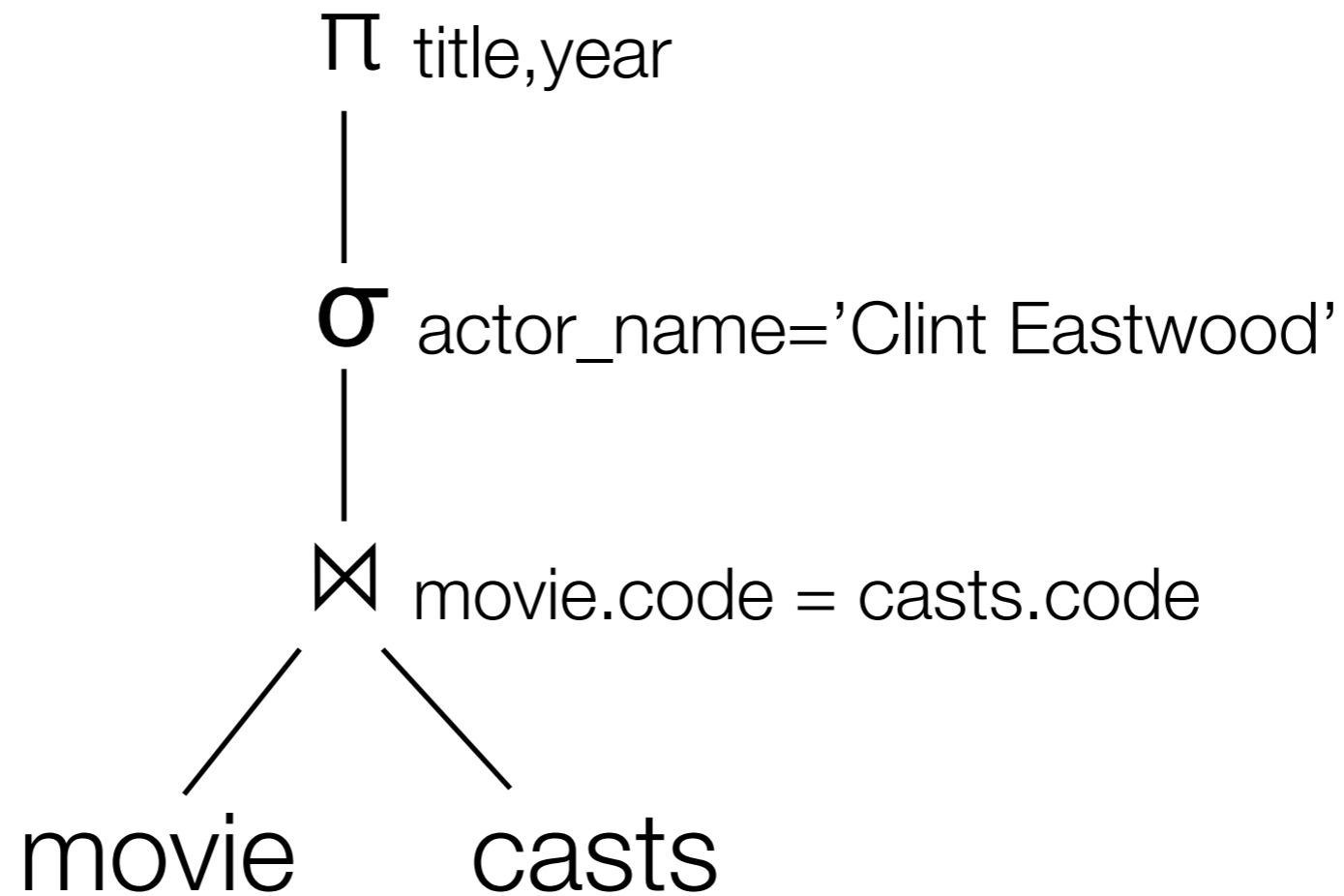
---



- The Parser inspects the SQL query, verifies its well-formedness, and converts it to the DBMS's internal format.
- The Planner considers many possible ways of implementing the query, and chooses one with a low estimated resource cost.
- The Execution Engine takes the query plan and loads the data from disk, performs the joins, filtering, aggregation etc, and returns the result to the user.

# Operator trees

---



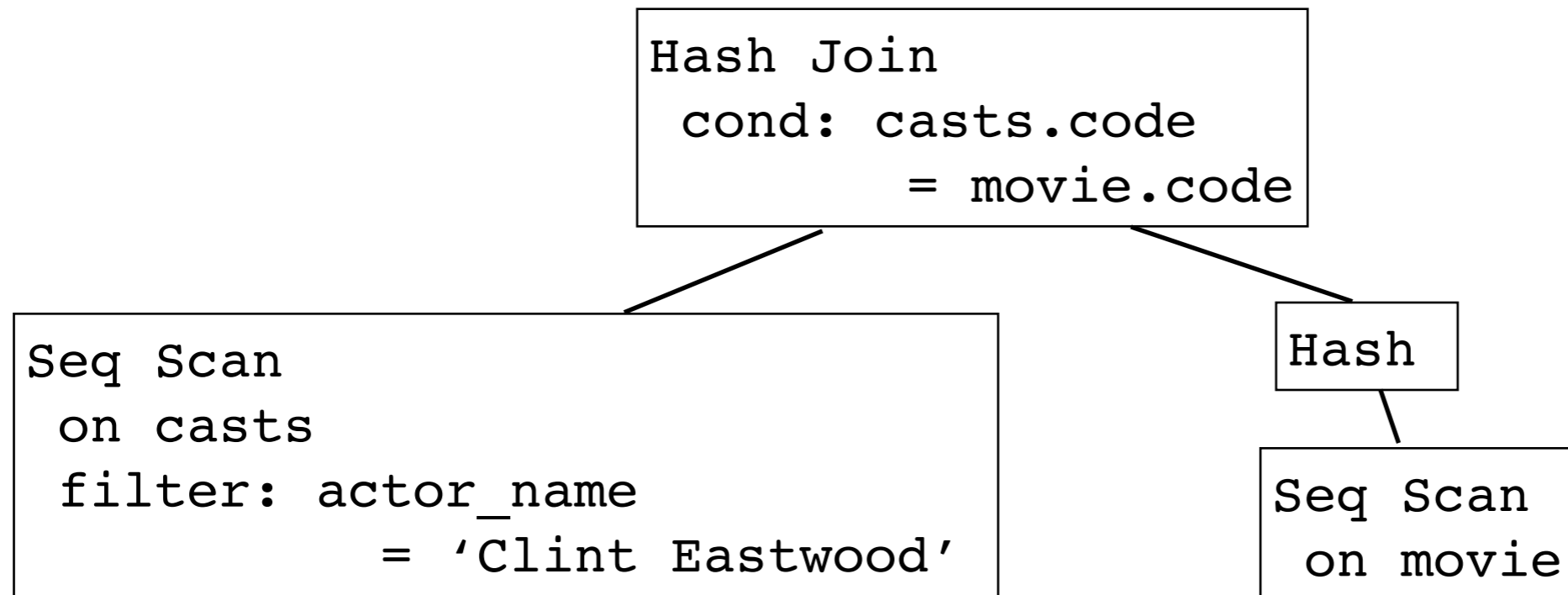
$\Pi$ title,year

( $\sigma$ actor\_name='Clint Eastwood')

(movie  $\bowtie$ movie.code = casts.code casts))

# Physical operator trees

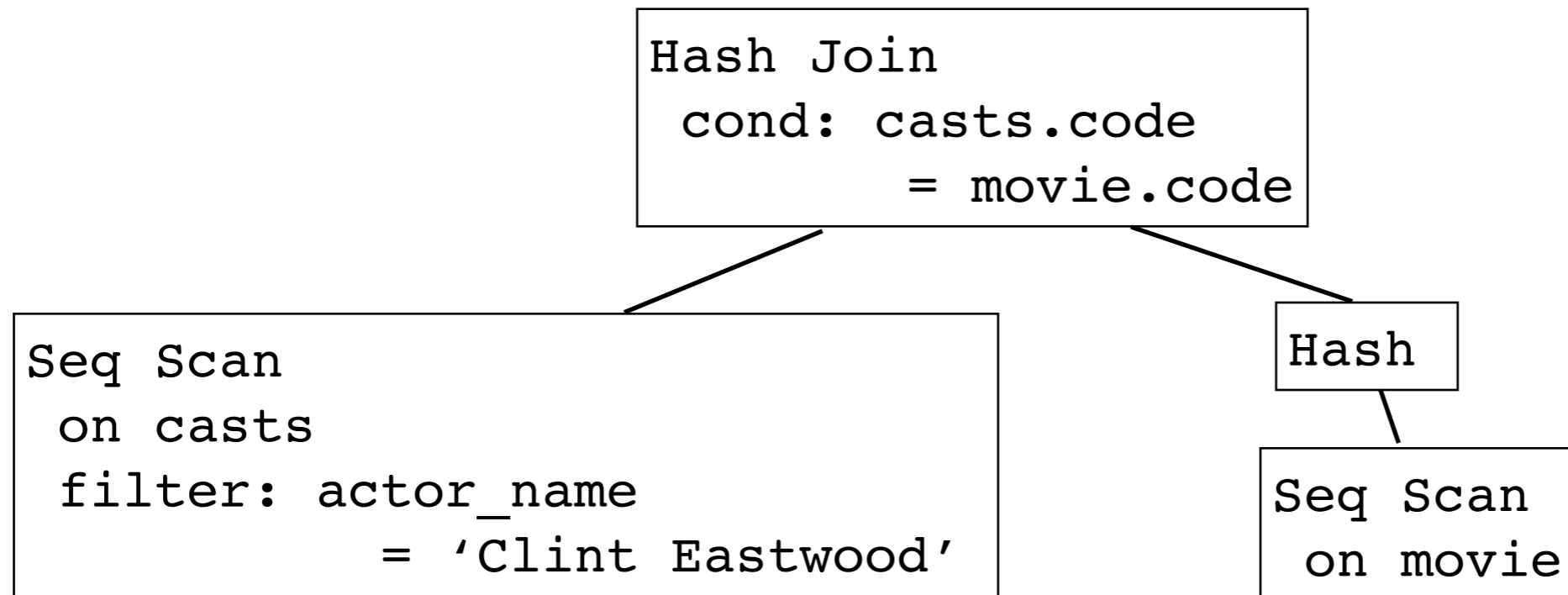
---



- A physical operator tree expresses a query plan.
- Each node is an algorithm implemented by the DBMS.
- The available physical operators vary with DBMS, though many are common.

# Physical operator trees (cont)

---



- Data flows from the leaves to the root of the tree.
- Data flow is usually demand driven.

Higher operators (closer to the root of the tree)  
request data from lower ones.

# EXPLAIN ANALYSE

---

## **EXPLAIN ANALYSE**

```
SELECT title, year
FROM    movie, casts
WHERE   movie.code = casts.code
         AND   actor_name = 'Clint Eastwood';
```

- With EXPLAIN we ask PostgreSQL what query plan it's using, including the expected cost.
- With ANALYSE we compare it with the predicted cost.
- EXPLAIN gives a flat representation of the physical operator tree.

```
Hash Join (cost=619.10..1535.79 rows=7 width=21)
          (actual time=32.554..55.041 rows=37 loops=1)
Hash Cond: ((casts.code)::text = (movie.code)::text)

-> Seq Scan on casts
      (cost=0.00..916.24 rows=7 width=5)
      (actual time=3.342..25.704 rows=38 loops=1)
Filter: ((actor_name)::text = 'Clint Eastwood'::text)

-> Hash (cost=475.71..475.71 rows=11471 width=26)
      (actual time=29.137..29.137 rows=11469 loops=1)
      -> Seq Scan on movie
            (cost=0.00..475.71 rows=11471 width=26)
            (actual time=0.012..10.975 rows=11471 loops=1)

Total runtime: 55.517 ms
(7 rows)
```

- We will work on decoding this next week!

# Join algorithms

---

- Joins lie at the heart of our queries.
- Joins are the relational operators which usually cost the most at runtime.
- There are several algorithms for performing joins,.
- The best algorithm to use depends on:
  - size of the data relative to memory
  - size of the tables relative to each other
  - whether we have indexes for the join attributes
  - statistical properties of the data
  - whether we're trying to minimize  
space usage, time usage, IO bandwidth etc..

# Nested loop join is the simplest.

---

$R \bowtie_c Q$

= do output := {}

**for each**  $r \in R$

**for each**  $q \in Q$

**if**  $rq \models c$  **then**

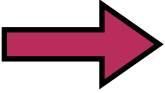
        output := output  $\cup$  {  $rq$  }

- $rq \models c$  means “In situation  $rq$ , statement  $c$  is true”
- We use two nested loops to iterate through the rows in each table.

FRUIT ⋈<sub>lid=fid</sub> COST

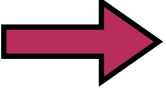
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

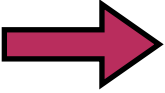
**OUTPUT**

fid	name	lid	price

FRUIT ⋈<sub>lid=fid</sub> COST

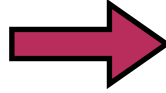
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

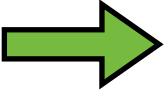
**OUTPUT**

fid	name	lid	price

FRUIT ⋈<sub>lid=fid</sub> COST

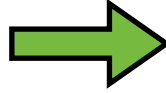
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

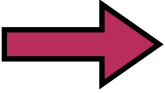
**OUTPUT**

fid	name	lid	price
12	apple	12	0.80

FRUIT ⋈<sub>lid=fid</sub> COST

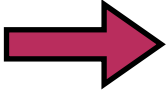
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

**OUTPUT**

fid	name	lid	price
12	apple	12	0.80

FRUIT ⋈<sub>lid=fid</sub> COST

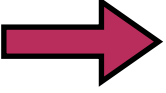
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

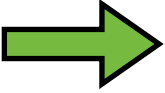
**OUTPUT**

fid	name	lid	price
12	apple	12	0.80

FRUIT ⋈<sub>lid=fid</sub> COST

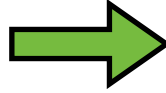
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

**OUTPUT**

fid	name	lid	price
12	apple	12	0.80
10	banana	10	1.10

FRUIT ⋈<sub>lid=fid</sub> COST

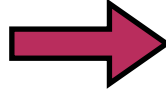
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

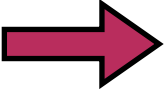
**OUTPUT**

fid	name	lid	price
12	apple	12	0.80
10	banana	10	1.10

FRUIT ⋈<sub>lid=fid</sub> COST

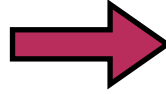
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

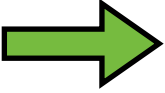
**OUTPUT**

fid	name	lid	price
12	apple	12	0.80
10	banana	10	1.10

FRUIT ⋈<sub>lid=fid</sub> COST

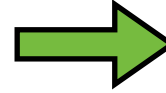
---

**FRUIT**



fid	name
12	apple
10	banana
5	cherry
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

**OUTPUT**

fid	name	lid	price
12	apple	12	0.80
10	banana	10	1.10
5	cherry	5	0.25

# etc etc etc

---

- Nested loop join is only worthwhile if:
  - .. both tables are small,
  - .. one is very small.
  - .. the more efficient algorithms can't be used on the data.
- If the size of the tables are M and N,  
the number of key comparisons needed is  $N * M$ .
- For our movie database:
  - size(movies) = **11,470**
  - size(casts) = **44,499**
  
  - $11,470 * 44,499 = 510,403,530$**  = 510 million = nightmare
- .. and this is only a single join!

# Merge join

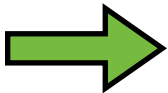
---

- Sort the tables on the join attributes (if needed)
- Scan both tables concurrently, matching up rows.
- Requires an ordering relation on the join attribute.
- If the tables have dense, ordered indexes on the join attributes, then we can use them instead of performing a physical sort.
- B<sup>+</sup>Trees are good because we can extract an in-order sequence of rows in an efficient manner. Hash tables don't support this..

FRUIT ⋈<sub>lid=fid</sub> COST

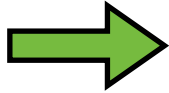
---

**FRUIT**



fid	name
5	cherry
10	banana
12	apple
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

**OUTPUT**

fid	name	lid	price
5	cherry	5	0.25

FRUIT ⋈<sub>lid=fid</sub> COST

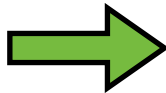
---

**FRUIT**



fid	name
5	cherry
10	banana
12	apple
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

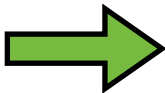
**OUTPUT**

fid	name	lid	price
5	cherry	5	0.25
10	banana	10	1.10

FRUIT ⋈<sub>lid=fid</sub> COST

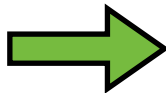
---

**FRUIT**



fid	name
5	cherry
10	banana
12	apple
27	durian

**COST**



lid	price
5	0.25
10	1.10
12	0.80
15	15.50

**OUTPUT**

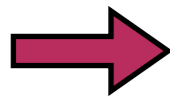
fid	name	lid	price
5	cherry	5	0.25
10	banana	10	1.10
12	apple	12	0.80

FRUIT ⋈<sub>lid=fid</sub> COST

---

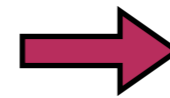
**FRUIT**

fid	name
5	cherry
10	banana
12	apple
27	durian



**COST**

lid	price
5	0.25
10	1.10
12	0.80
15	15.50



**OUTPUT**

fid	name	lid	price
5	cherry	5	0.25
10	banana	10	1.10
12	apple	12	0.80

# Merge join is made of WIN.

---

- When a row doesn't match:  
advance the lowest pointer to the first value  $\geq$  the highest one.
- There are some slight complications when we allow repeated values in the columns.

Handle this case by sorting rows into “packets” with the same value, use a single loop over the members of the packet.

- The *output* of a merge join is also sorted, so it can be fed directly to a subsequent merge join operator.
- If the tables fit in internal memory we can use quicksort which has time complexity  $O(n \cdot \log n)$

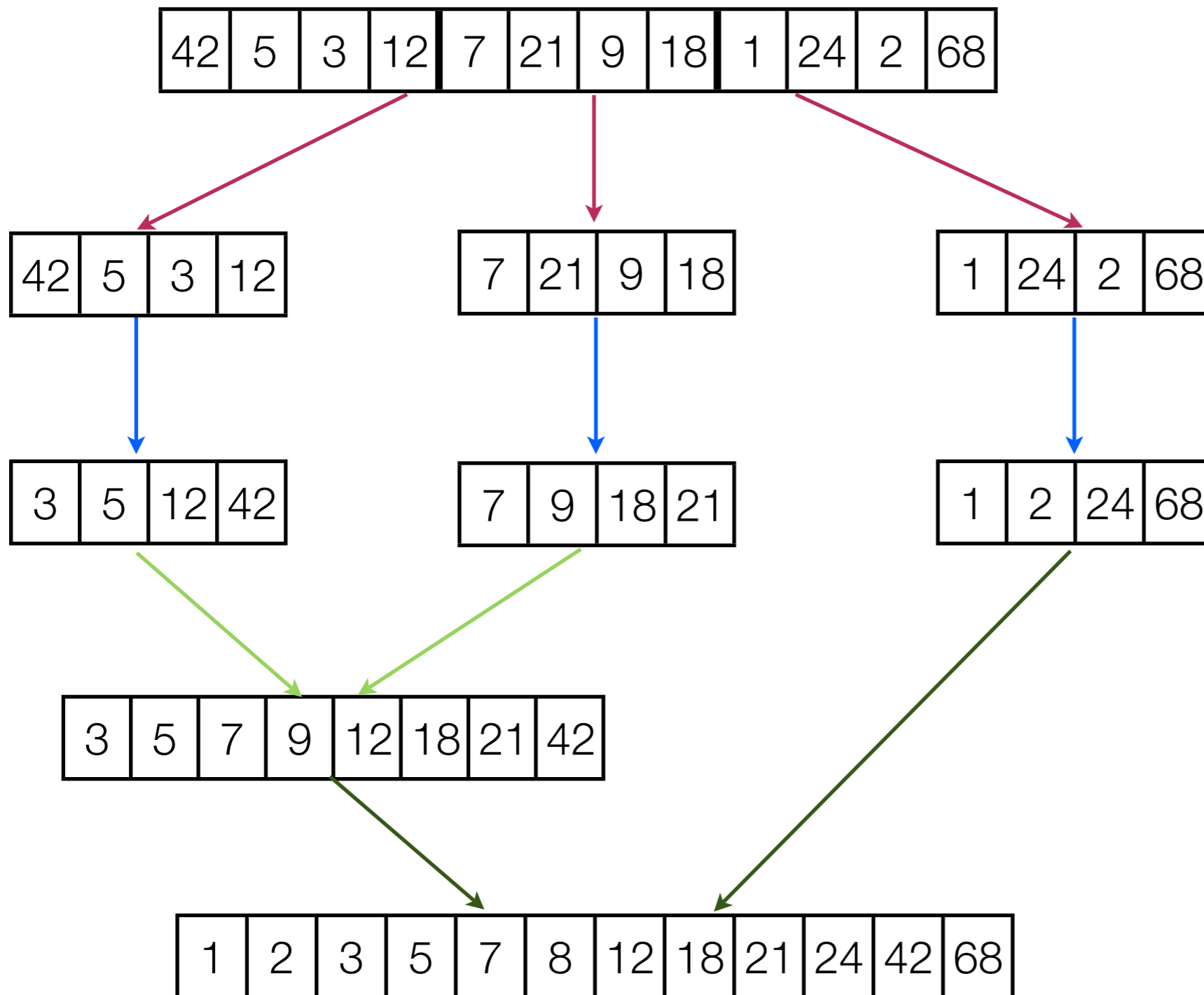
# Merge join is good for very large tables.

---

- Merge join can work on tables that are too big to fit in internal memory.
- We sort the tables into temporary files, then perform the join while streaming both tables from disk.
- A good algorithm for sorting large tables is “merge sort”:
  - partition each table into sections  
(usually each section is as much as will fit in memory at a time)
  - sort each section individually.
  - merge sorted sections.
- We can sort each partition with a fast in-memory algorithm.
- For very big files we may need several passes..

# Merge sort example.

---



partition

in-memory sort

merge

merge