Introduction to Database Systems
I2DBS – Spring 2023

- Week 9:
- Indexes (Recall)
- Query Processing

Jorge-Arnulfo Quiané-Ruiz

Readings:

PDBM 13.1

Profile of the Week

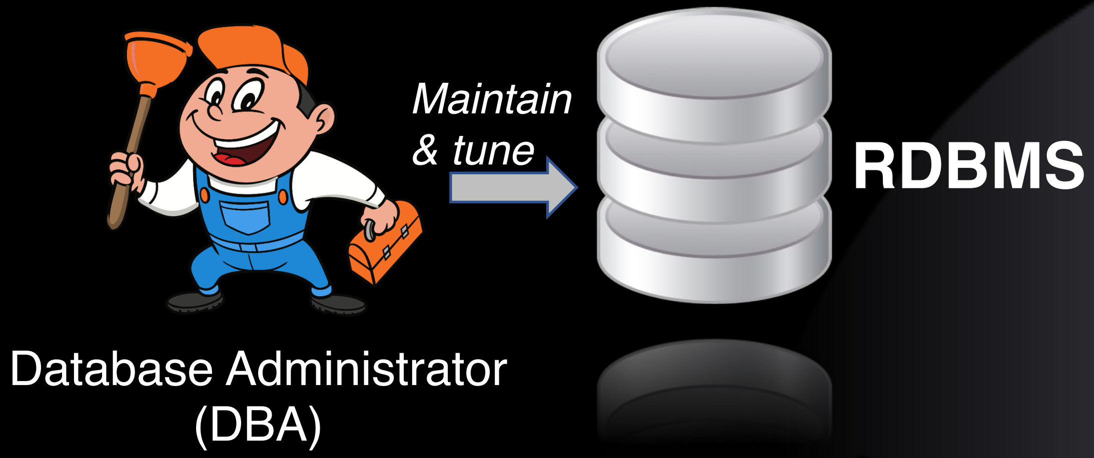
David DeWitt

Inventor of Hybrid Hash Join

- **1948:** USA
- **1976:** PhD in CS from the University of Michigan
- **1976:** founded the Wisconsin Database Group
- **1984:** Invented Hybrid Hash Join
- **1995:** SIGMOD Edgar F. Codd Innovations Award
- **2009:** ACM Software System Award

The Dewitt Clause

End-user license agreement provision that prohibits researchers and scientists from explicitly using the names of their systems in academic papers.



Indexes (Recall)

Readings:

PDBM 12

Recap: Indexing

- **Indexes are data structures that facilitate access to data from disk**
  - … if conditions are a prefix of indexed attributes
  - Clustered indexes store tuples that match a range condition together
  - Some queries can be answered looking only at the index (a covering index for query)
  - Indexes slow down updates and insertions

- **The choice of whether to use an index is made by the DBMS for every instance of a query**
  - May depend on query parameters
  - Don't have to name indexes when writing queries

Processing Simple Selections

- **Point and range queries on the attribute(s) of a clustered index are almost always best performed using an index scan**
- **Unclustered indexes should only be used with high selectivity queries**
- **Exception: Covering index is good for any selectivity**
- **If no index exists, a full table scan is required!**
- **If no “good” index exists, a full table scan is preferred!**

Processing Complex Selections

- **We consider the conjunction (”and”) of equality and range conditions.**
- **No relevant index: Full table scan**
- **One relevant index:**
  - Highly selective: Use that index
  - If not: Full table scan
- **Multiple relevant indexes:**
  - One is highly selective: Use that index
  - No single condition matching an index is highly selective: Can “intersect” the returned sets

Using a Highly Selective Index

- **Basic idea:**
  - Retrieve all matching tuples (few)
  - Filter according to remaining conditions

- **If index is clustered or covering: Retrieving tuples is particularly efficient, and the index does not need to be highly selective.**

Using Several Less Selective Indexes

- **For several conditions C1, C2,... matched by indexes:**
  - Retrieve the addresses Ri of tuples matching Ci.
  - The addresses are in the index leaves!
  - Compute the intersection R = R1 ∩ R2 ∩ ...
  - Retrieve the tuples in R from disk (in sorted order)

- **Remaining problem:**
  - How can we estimate the selectivity of a condition? Of a combination of conditions?
  - Use some stats and probabilistic assumptions…

Example

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

- **Examples of strategies:**

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Example – Variant 1

Available Indexes:

- CREATE INDEX yearIdx ON Movie(year)
- CREATE INDEX studIdx ON Movie(studioName)

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

- **Examples of strategies:**

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Which strategies are possible and which index would be used?

Example – Variant 2

Available Indexes:

- CREATE INDEX yearIdx
  ON Movie(year)
- CREATE INDEX yearStudIdx
  ON Movie(year, studioName)

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

- **Examples of strategies:**

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

Which strategies are possible and which index would be used?

Example – Variant 3

Available Indexes:

- CREATE INDEX idIdx ON Movie(id)
- CREATE INDEX titleIdx ON Movie(title)

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

- **Examples of strategies:**
1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine two indexes to identify rows fulfilling both conditions.
5. Use one composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 and their titles in a composite covering index.

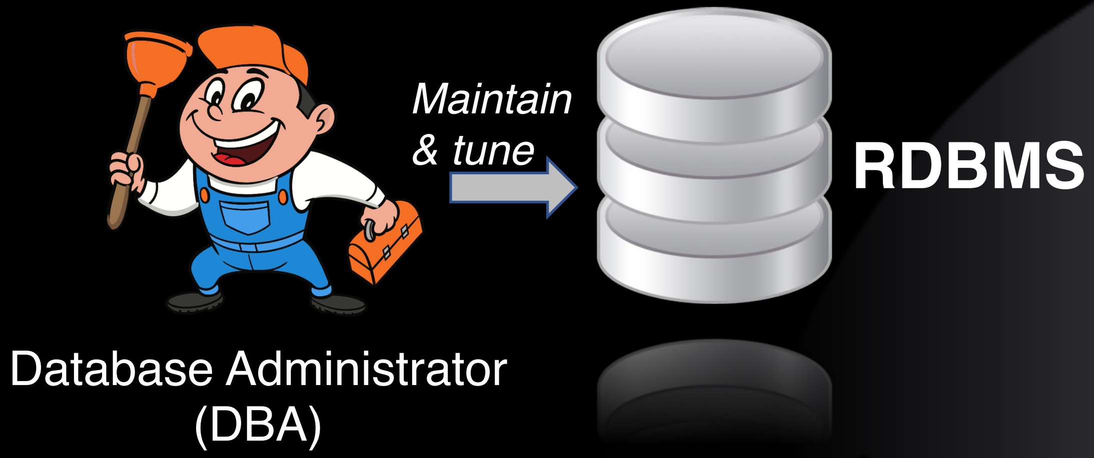
Which strategies are possible and which index would be used?

Processing Complex Selections Revisited

- **We have considered the conjunction (”and”) of a number of equality and range conditions.**

- **What about disjunctive (”or”) selections?**
  - One full table scan

    OR
  - Multiple “and” queries



Query Processing

Readings:

PDBM 12

Query Evaluation in a Nutshell

- **SQL rewritten to (extended) relational algebra**
- **The building blocks in DBMS query evaluation are algorithms that implement relational algebra operations.**
  - Join is the most important one!
- **May be based on:**
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- **The DBMS optimizer knows the characteristics of each approach, and attempts to use the best one in a given setting**

Join Evaluation in a Nutshell

- **Join is the most important operation!**
- **May be based on:**
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- **We consider a simple join:**

  R JOIN S ON S.ID = R.ID

  - Extends to more complex joins in a straightforward way

Nested Loops Join

- **The following basic algorithm can be used for any join:**

```
for each tuple in R
    for each tuple in S
        if r.ID = s.ID
        then output (r, s)
```

- **If the join condition is complex/broad, sometimes this is the only/best choice**

  R JOIN S ON S.ID <> R.ID

- **See animation example on LearnIT**

Role of Index in Nested Loops Join

- **If there is an index that matches the join condition, the following algorithm can be considered:**

  For each tuple in R

  use the index to locate matching tuples in S

- **See animation example on LearnIT**
- **Good if |R| is small compared to |S|**
- **If many tuples match each tuple, a clustered or covering index is preferable.**

Example

```
SELECT *
FROM Movie M, Producer P
WHERE M.year=2015
  AND P.birthdate<'1940-01-01'
  AND M.producer = P.id;
```

- **Some possible strategies:**
  1. Use index to find 2015 tuples, use index to find matching tuples in Producer.
  2. Use index to find producers born before 1940, use index to find matching movies.
  3. NL join Movie and Producer, then filter.

Problem Session

- **What would be good indexes for this query?**

```
SELECT A.street, A.streetno
FROM person P
   JOIN address A ON A.person_id=P.id
WHERE P.lastname='Bohr’
  AND P.firstnames LIKE 'Niels%';
```

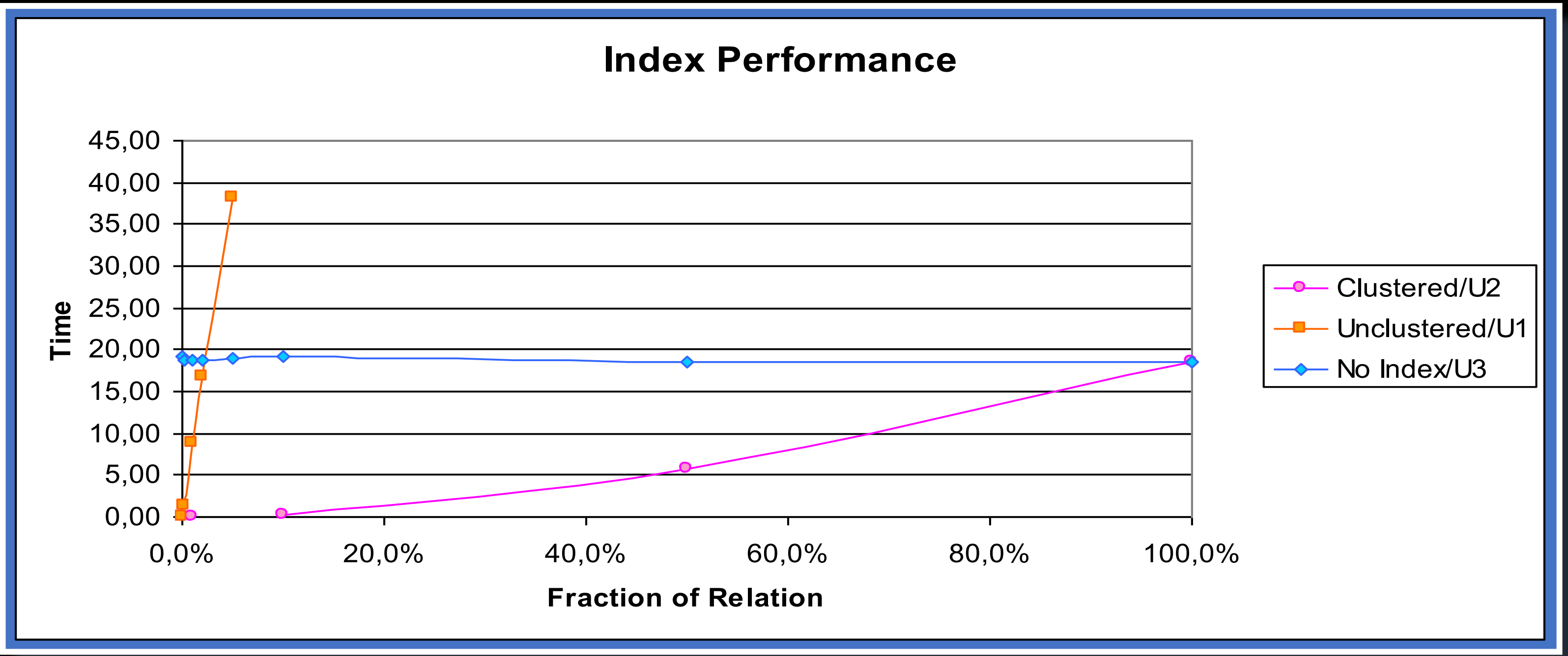
Merge Join

- **Consider R JOIN S on R.ID = S.ID**
  - Step 0: Sort R and S on ID
  - Step 1: Merge the sorted R and S
  - See animation example on LearnIT

- **Cost:**
  - If already sorted: $O(|R| + |S|)$
  - Can we do better?
  - If not sorted:
    $O(|R|\log|R| + |S|\log|S| + |R| + |S|)$

Role of Indexes in Merge Joins

- **Indexes can be used to read data in sorted order**
- **When is this a win?**
  - Index is clustered
  - Index is covering
- **When is this a loss?**
  - Index is unclustered



Hash Join

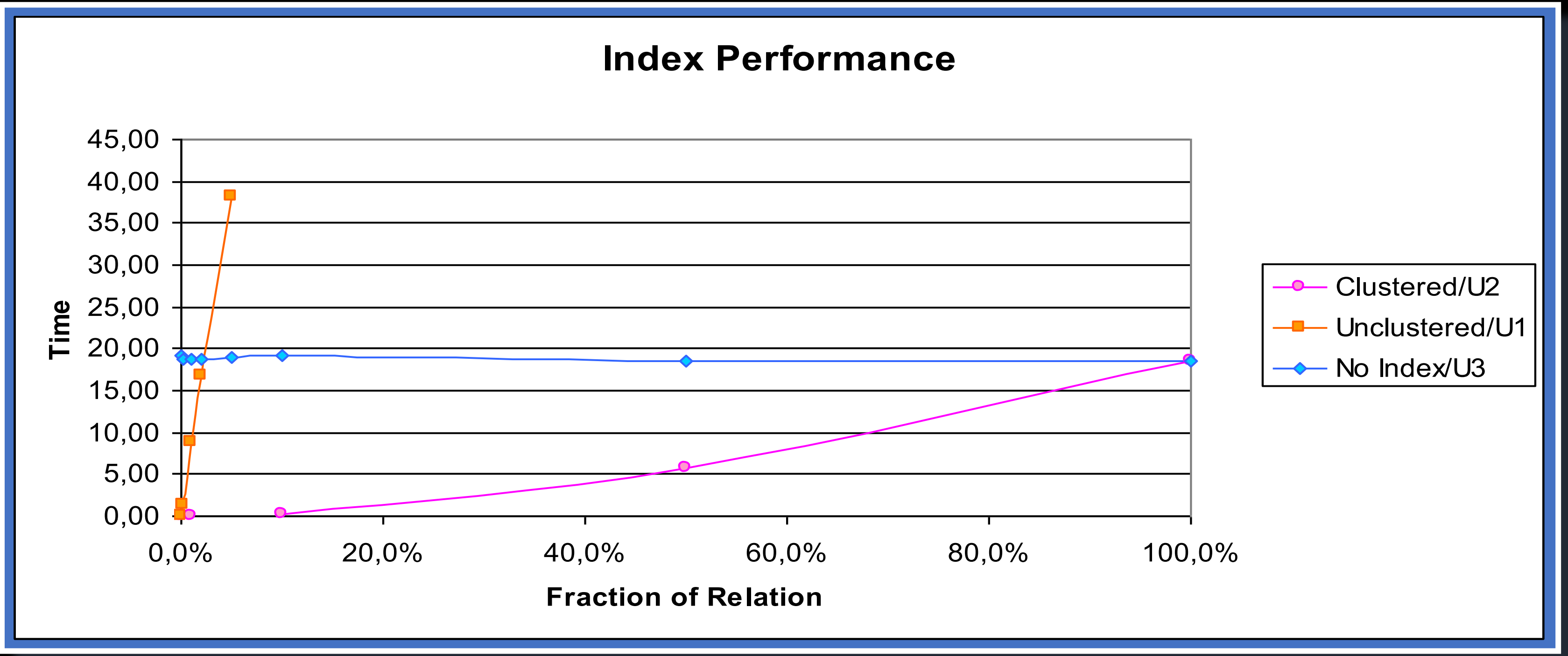
- **Consider R JOIN S on R.ID = S.ID**
  - Best if S fits in RAM

  Step 0: Create a good hash function for ID
  Step 1: Create a hash table for S in memory
  Step 2: Scan R and look for matching tuples in the hash table
  - See animation example on LearnIT

- **Cost: O(IRI + ISI)**
  - Can we do better?
  - What is S does not fit in RAM?

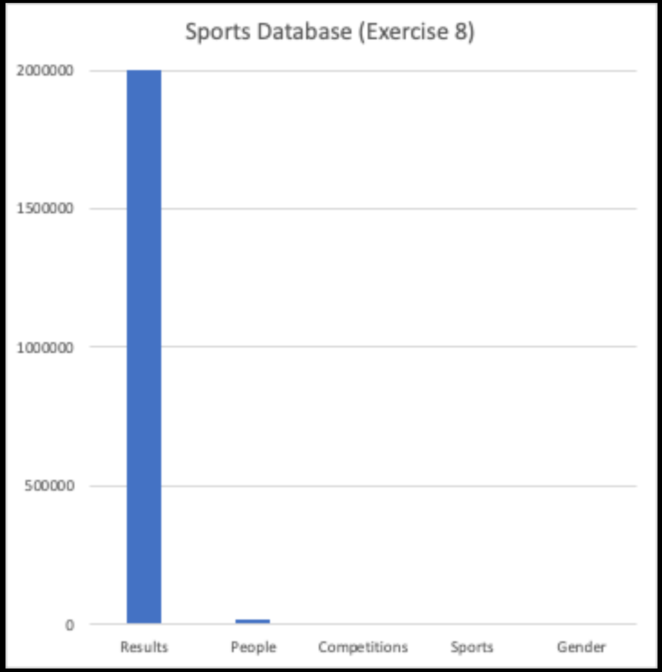
Role of Indexes in Hash Joins

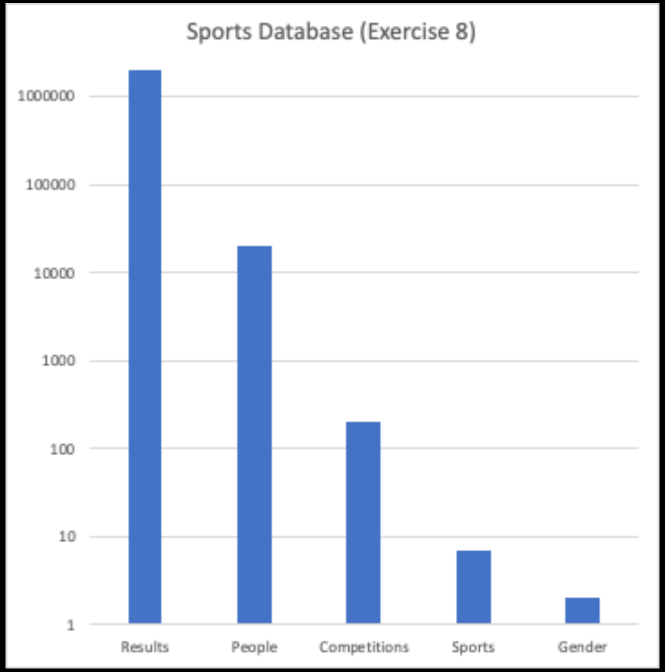
- **Hash joins read all the relations by default**
- **How can indexes be useful?**
  - Apply to non-join conditions – before join
  - Index is covering



Comparison of Join Algorithms

- **Nested loops join:**
  - Very costly O(|R|*|S|)
  - Works for any condition → sometimes only option
- **Merge join:**
  - Works well if data is well clustered
  - Works well if relations are large and similar in size
- **Hash join:**
  - Works well if one relation is small
  - Is that often the case?





Grouping Operations

- **Many operations are based on grouping records (of one or more relations) according to the values of some attribute(s):**
  - Join (group by join attributes)
  - Group by and aggregation (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)

- **Most database systems implement such grouping efficiently using sorting or hashing**

Partitioning of Tables

- **A table might be a performance bottleneck**
  - If it is heavily used, causing locking contention (more on this later in course)
  - If its index is deep (table has many rows or search key is wide), increasing I/O
  - If rows are wide, increasing I/O
- **Table partitioning might be a solution to this problem.**

Horizontal Partitioning

- **If accesses are confined to disjoint subsets of rows, partition table into smaller tables containing the subsets**
  - Geographically, organizationally, active/inactive
- **Advantages:**
  - Spreads users out and reduces contention
  - Rows in a typical result set are concentrated in fewer pages
- **Disadvantages:**
  - Added complexity
  - Difficult to handle queries over all tables

Vertical Partitioning

- **Split columns into two subsets, replicate key**
- **Useful when table has many columns and**
  - it is possible to distinguish between frequently and infrequently accessed columns
  - different queries use different subsets of columns
- **Example: Employee table**
  - Columns related to compensation (tax, benefits, salary) split from columns related to job (department, projects, skills).
- **DBMS trend (for analytics):**
  - Column stores, with full vertical partitioning.
  - More on this next week.

Takeaways

- **The performance difference between well-tuned and poorly-tuned applications can be massive!**
- **The DBMS does its best to optimize queries, but sometimes it needs help!**
  - Query tuning – rewrite as joins or non-correlated subqueries
  - Indexes – solve 90+% of all other performance problems
- **If that is not sufficient…**
  - Materialized views / Partitioning / Denormalization
  - Beyond the scope of this course!

What is next?

Next Lecture:

- RDBMS Implementation
- Main Memory DBMSes