QUERY PROCESSING IN A RDBMS

University of Waterloo

List of Slides

Services Provided by a DBMS

- Data Independence

  $\Rightarrow$ transparent mapping of relations to *data structures*
  AND queries to low-level *execution plans*

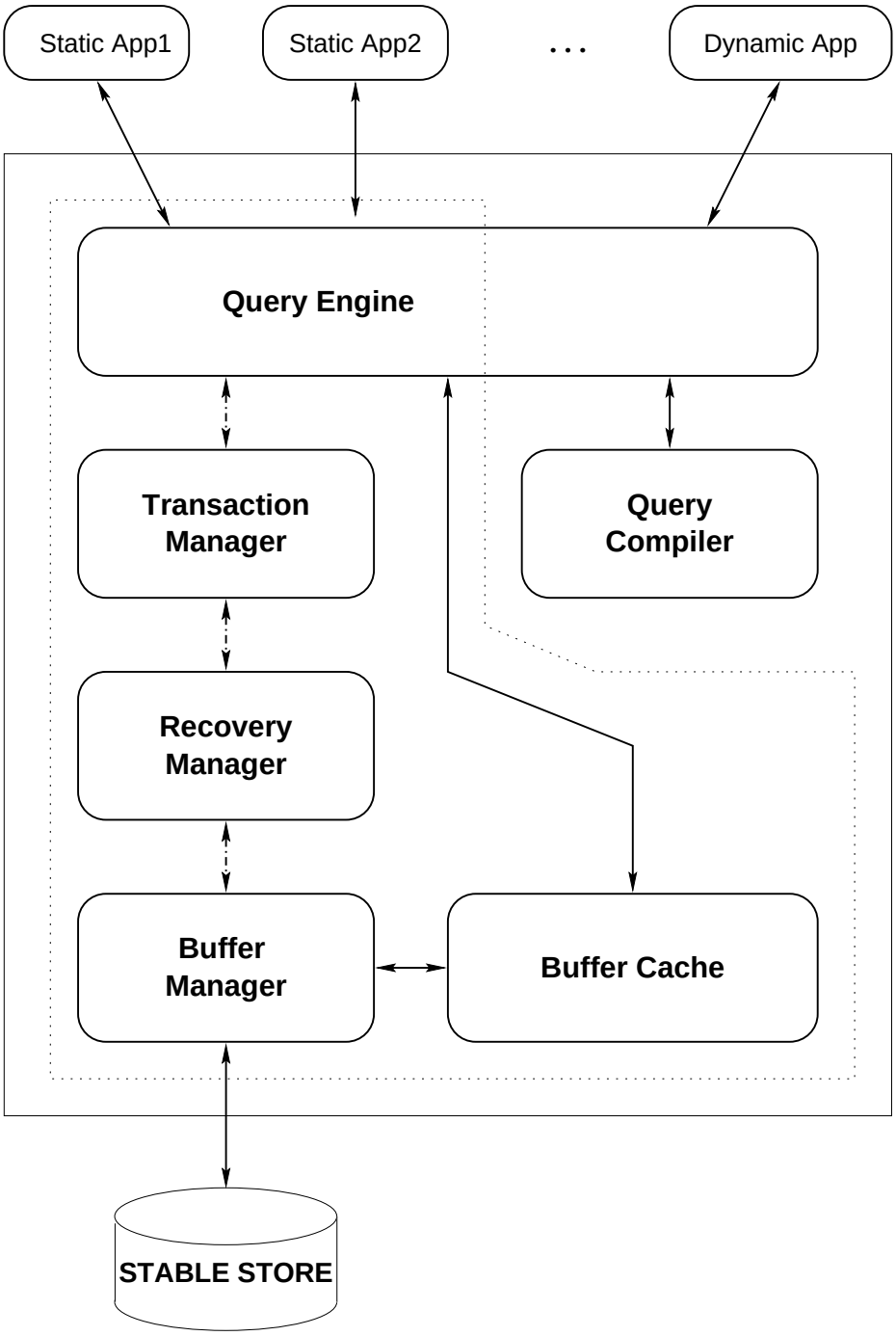
- Concurrency control

  $\Rightarrow$ transparent concurrency between DB applications
  virtual single-user system behavior

- Recovery

  $\Rightarrow$ guarantees that data won't be lost

RDBMS Structure



Physical Data Storage

Database systems store data on permanent media:

- disks (most common)
- tapes (archives, large quantities of data)

The essential requirement is that the medium is **not volatile** (except due to catastrophic failures).

$\Rightarrow$ major implication on DBMS design

High performance systems:

$\Rightarrow$ battery backed RAM would do (but it is too expensive)
$\Rightarrow$ main-memory databases (e.g., a phone switch)

Disks

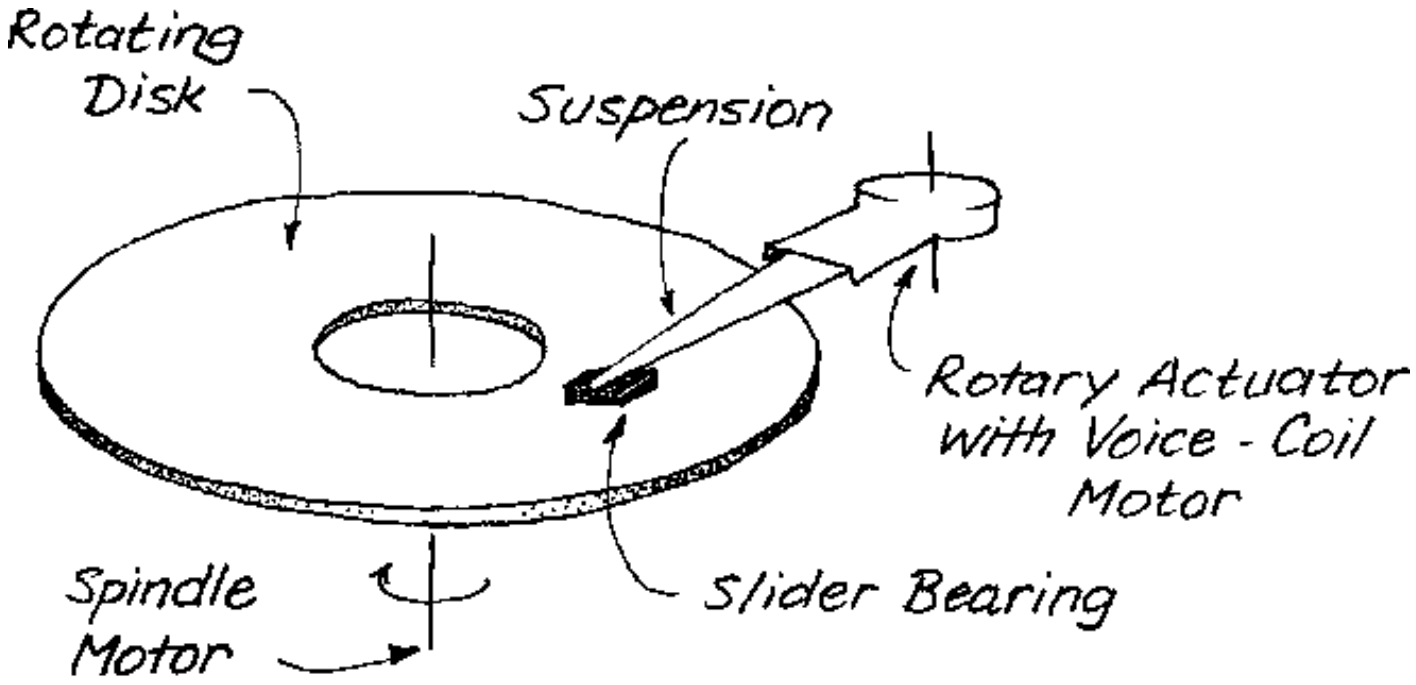


Figure 1: Major electromechanical components of a conventional high-performance rigid disk drive.

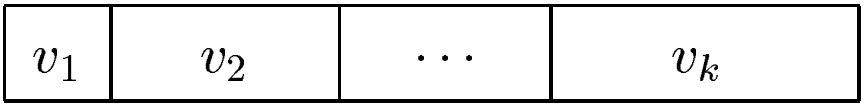
- seek time (to find the right track: 2-20ms)
- rotational delay (to get the right sector: 8-17ms)
- data transfer (1ms, 1-5MB/s)

but brings in data in **pages** (e.g., 512 bytes)

Tuples vs. Records

IDEA: logical tuples map to *records*.

- fixed record type:

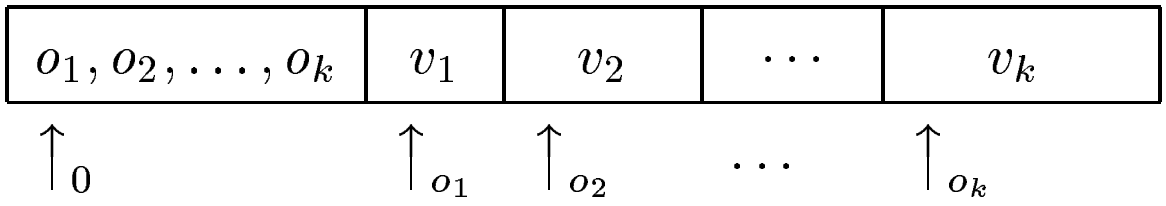


$\Rightarrow$ essentially a fixed array of bytes

$\Rightarrow$ the **length** (in bytes) of every field is fixed
(and stored in the database catalogs)

- variable length records

use a directory of offsets stored at the
beginning of the record to locate the fields:



$\Rightarrow$ may be combined with fixed field record(s)
or with records-in-page layout

What about Pages?

- how do records fit in pages (disk blocks)?
  1. packed (seldom used: records move too often)
  2. unpacked + a directory (works for variable length records too)
- records can be located using **record id (rid)**:
  - $\Rightarrow$ rid tells us what page (disk block) and what offset the beginning of a record is located at.
  - $\Rightarrow$ used by indices to locate records fast

Records and Files

IDEA: logical relations map to *files*

- many different ways of organizing files
  - $\Rightarrow$ heap (unordered, as a linked-list or with page directory)
  - $\Rightarrow$ sequential (ordered by a field)
  - $\Rightarrow$ directory-like (1 level ISAM)
  - $\Rightarrow$ . . .
- provides an interface for query processor:
  - $\Rightarrow$ an uniform query interface (to list all records) (e.g., an *iterator protocol*)
  - $\Rightarrow$ an insertion and deletion of a record
  - $\Rightarrow$ specialized ways to locate records (optional)
- different computational properties (search/modification speed tradeoffs)

Clustering

Idea: To minimize the disk I/O delays we try to store related records close on the disk.

- consecutively scanned records of a single relation on consecutive disk blocks (often quite sufficient).
- storing records often retrieved together on consecutive pages (e.g., customer and his/her account(s)).
  - – very complex optimization problem
  - – often, ad-hoc solutions used

$\Rightarrow$ depends on what queries are most frequent

Indexing in Databases

- can we always do relation scans? **NO:**

  $\Rightarrow$ looking for a record among 100k records: 120s
  $\Rightarrow$ with a (tree-like) index: less than 1s

- looking up a few records is very common

  $\Rightarrow$ support simple search condition (point, range)
  $\Rightarrow$ additional “search” structures on top of files
  $\Rightarrow$ we trade space for execution time

- defines **search attribute(s)** for a relation:

  given values of the search attributes, we can find the matching tuples really really fast . . .

Primary vs. Secondary Indices

- in general, the **data entries** are stored
  1. directly in the index:

     $\Rightarrow$ a data record with key value k

     $\Rightarrow$ a **primary index** (often sparse)
  2. indirectly, using a foreign rid:

     $\Rightarrow$ a key value k and the matching rid pair

     $\Rightarrow$ a **secondary index** (usually dense)
- in case of duplicates:

  $\Rightarrow$ overflow chains
- this choices are (almost) orthogonal to the choice of the indexing method.

Tree-based Indices

- **Idea:** tree based data structure allow search in $\log n$

  $\Rightarrow$ how to make them work on disk?

  $\Rightarrow$ how to make them dynamic?

- B-trees:

  $\Rightarrow$ are widely-used index structures.

  $\Rightarrow$ are dynamic: they grow and shrink easily.

  $\Rightarrow$ have two parts: index blocks and data blocks.

  $\Rightarrow$ index and data blocks are kept at least half full.

- many other variants: R-tree, R*-tree, HB-tree, . . .

- even B-trees come in several flavours;

  $\Rightarrow$ we will discuss B+ trees.

Another Method: Hashing

- **Idea:** use a hash function to map records to pages:

  $\Rightarrow$ the hash function of the searched-for value gives the page where the records are located.

  $\Rightarrow$ with a *good* hash function almost $O(1)$

  $\Rightarrow$ very cheap to construct (intermediate results)

  $\Rightarrow$ how do we make this dynamic on a disk?

- Various dynamic schemes:

  $\Rightarrow$ Extensible Hashing, **Linear Hashing**, . . .

Queries to Plans

Idea: “queries = functions over a universe of relations”.

$\{\theta : D, \theta \models \varphi\}$ is implemented as $F_\varphi(r_1, \ldots, r_k)$

- universe $\mathcal{U}$: finite relations over DOM
- and **relational operations:**
  - $\Rightarrow$ Projection: $\pi_V : \mathcal{U} \to \mathcal{U}$,
  - $\Rightarrow$ Selection: $\sigma_\varphi : \mathcal{U} \to \mathcal{U}$
  - $\Rightarrow$ Product: $\times : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$
  - $\Rightarrow$ Union: $\cup : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$
  - $\Rightarrow$ Difference: $- : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$

operators are easy to implement and **can be composed**

Calculus-to-Algebra Translation

The translation is a simple recursive procedure:

$$
\begin{aligned}
\text{Trans}(R(x_{i_1}, \ldots, x_{i_k})) &= R \\
\text{Trans}(Q_1 \wedge Q_2) &= \text{Trans}(Q_1) \times \text{Trans}(Q_2) \\
\text{Trans}(Q \wedge x_i = x_j) &= \sigma_{x_i = x_j}(\text{Trans}(Q)) \\
\text{Trans}(\exists x_i . Q) &= \pi_{FV(Q) - \{x_i\}}(\text{Trans}(Q)) \\
\text{Trans}(Q_2 \vee Q_3) &= \text{Trans}(Q_1) \cup \text{Trans}(Q_2) \\
\text{Trans}(Q_2 \wedge \neg Q_3) &= \text{Trans}(Q_1) - \text{Trans}(Q_2)
\end{aligned}
$$

where Q_1 and Q_2 have disjoint sets of free variables and Q_2 and Q_3 are union compatible.

Theorem [Codd]:

For every (safe) relational calculus query there is an equivalent RA expression

Duplicate Operations

- Projection and duplicate elimination

  $\Rightarrow$ (set) projection (π) is usually split to

  1. a duplicate preserving projection (π)
  2. a duplicate elimination operator (ϵ)

- Aggregates, counting, etc.

  $\Rightarrow$ additional operator $\mathrm{Agg}_G^{f_1,\ldots,f_k}(R)$

  groups by columns in G

  applies aggregates f_i (new columns in result).

- Duplicate versions of standard operators

  $\Rightarrow$ Product, Selection (always preserve duplication)

  $\Rightarrow$ Duplicate preserving Union and Difference

Implementation of RA Operators

- every of the Relational Algebra operators can be implemented in several ways
  - $\Rightarrow$ not always clear which is the best choice
  - $\Rightarrow$ we implement as many as possible (so we can pick depending on the particular query and database instance).
- the operators are composed using an *iterator protocol*.
- in practice, the operations are often decomposed to more primitive operations (e.g., retrieve a pointer to a record and extract a field from previously retrieved record).

Atomic Relations

We use access methods to gain access to the stored data:

- if an index $R_{\mathrm{index}}(x)$ (where x is the *search attribute*) is available we replace a subquery of the form

  $$\sigma_{x=c}(R)$$

  with accessing $R_{\mathrm{index}}(x)$ directly,
- Otherwise: check all file blocks holding tuples for R.

Joins

- THE most studied operation of relational algebra; There are many other ways to perform a join.
  1. The *Nested Loop* Join

     ```
     for t in R  do for u in S do
             if C(t,u) then output (tu)
     ```

     $\Rightarrow$ with the optional use of indices on **S**
  2. The *Sort-Merge* Join

     sort the tuples of **R** and of **R** on the common values, then merge the sorted relations.
  3. The *Hash* Join

     hash each tuple of **R** and of **S** to "buckets" by applying a hash function to columns involved in the join condition. Within each bucket, look for tuples with the matching values.
- the *cost* of the join depends on the chosen method

Duplicates and Aggregates

How do we eliminate duplicates in results of operations?
How do we group tuples for aggregation?

Similar solution:

1. sort the result and then eliminate duplicates/aggregate
2. hash the result and do the same

$\Rightarrow$ often an index (e.g., a B+ tree) can be used to avoid the sorting/hashing phase

The rest of the lot

- we assume a natural implementation for selection, duplicate-preserving projection, and duplicate preserving union.
- set difference can be evaluated similarly to a join.
- additional operations:
  - $\Rightarrow$ sorts (used for Sort-Merge Join, Aggregation, and Duplicate Elimination). Uses an *external sort algorithm* (essentially a merge-sort adopted for disk)
  - $\Rightarrow$ temporary store (to avoid recomputation of subqueries; can be inserted anywhere in the query plan)
  - $\Rightarrow$ ...

Summary

Physical operator trees (*query plans*) *implement*
Relational Algebra queries/expressions
(and, in turn, Relational Calculus/SQL).

Naive implementation seems *easy*

$\Rightarrow$ . . . but to make this work well is tricky:

- how to avoid reading the same data?
- preferable “query plans”?
- use of advanced *data structures*
  that work well with disk??
- data from multiple sources?
  information integration