Chapter 5: Physical Database Design and Performance

Introduction

• How are tables stored on the disk?
• Main (only?) issue is performance

• Minimize disk access
• Logical design should NOT change!
• Many decisions should be deferred until implementation/maintenance
• Highly DBMS dependent
Physical Design Process

**Input**
- Normalized relations
- Volume estimates
- Attribute definitions
- Response time expectations
- Data security needs
- Backup/recovery needs
- Integrity expectations
- DBMS technology used

**Decisions**
- Attribute data types
- Fields and physical records descriptions (doesn’t always match logical design)
- File organizations
- Indexes and database architectures
- Query optimization

**Figure 5-1 Composite usage map**
(Pine Valley Furniture Company)

Data (transaction) volumes
Memory Hierarchy

• Many different kinds of memory
  – Primary (RAM)
  – Secondary (Hard Drives, now Flash/SSD)
  – Tertiary – Removable, Tapes

• Tradeoff price vs. speed (for same size)
  – So now we have more slower kinds of memory

• Also, qualitative difference
  – Primary is volatile
  – Secondary is permanent, non-removable
  – Tertiary is removable
Hard Disks

• Most databases reside in hard disks

• Hard disk issues
  – Info is transferred in *pages*
  – Slight difference in access time to reach each page, but considered the same
  – Speed: seek time, transfer rate (about 100 times slower than RAM)

• The fewer the pages I read/write, the faster my database

---

This figure shows how data are recorded on magnetic disks.

Figure: Magnetic Disk Components
Fields: Choosing Data Types

- CHAR—fixed-length character
- VARCHAR—variable-length character (memo)
- LONG—large number
- NUMBER—positive/negative number
- INTEGER—positive/negative whole number
- DATE—actual date
- BLOB—binary large object (good for graphics, sound clips, etc.)

Fields: Coding

- Uses a reference instead of an actual value, and puts the actual value in another table
- In a way, transforming an attribute into an entity in an ER diagram
- Saves space but needs additional lookup
- Can improve integrity

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alex</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>Ela</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>Jason</td>
<td>M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Male</td>
</tr>
<tr>
<td>F</td>
<td>Female</td>
</tr>
</tbody>
</table>
Physical Records

• Physical Record: A group of fields stored in adjacent memory locations and retrieved together as a unit (~row)
• Page: The amount of data read or written in one I/O operation (device dependent)
• Blocking Factor: The number of physical records per page
  – May vary!

Example Pages

<table>
<thead>
<tr>
<th>Staff#</th>
<th>Position</th>
<th>Branch#</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Assistant</td>
<td>1</td>
</tr>
<tr>
<td>M1</td>
<td>Manager</td>
<td>1</td>
</tr>
<tr>
<td>A2</td>
<td>Assistant</td>
<td>2</td>
</tr>
<tr>
<td>M2</td>
<td>Manager</td>
<td>2</td>
</tr>
</tbody>
</table>
**Denormalization**

- DON’T!
- Transforming normalized relations into denormalized ones
- Possible benefits: fewer joins, faster
- Costs: confusing!, wasted space, integrity threats

**Denormalization**

- Common denormalization opportunities
  - One-to-one relationship (Fig. 5-3)
  - Many-to-many relationship with attributes (Fig. 5-4)
Figure 5-3 A possible denormalization situation: two entities with one-to-one relationship

Normalized relations:

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>APPLICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student ID</td>
<td>Application ID</td>
</tr>
<tr>
<td>Campus Address</td>
<td>Application Date</td>
</tr>
</tbody>
</table>

Denormalized relation:

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>APPLICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student ID</td>
<td>Application ID</td>
</tr>
<tr>
<td>Campus Address</td>
<td>Application Date</td>
</tr>
</tbody>
</table>

Null description possible

Figure 5-4 A possible denormalization situation: a many-to-many relationship with nonkey attributes

Normalized relations:

<table>
<thead>
<tr>
<th>VENDOR</th>
<th>PRICE QUOTE</th>
<th>ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor ID</td>
<td>Price</td>
<td>Item ID</td>
</tr>
<tr>
<td>Address</td>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>Contact Name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Denormalized relations:

<table>
<thead>
<tr>
<th>VENDOR</th>
<th>PRICE QUOTE</th>
<th>ITEM QUOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor ID</td>
<td>Item ID</td>
<td>Description</td>
</tr>
<tr>
<td>Address</td>
<td>Item ID</td>
<td>Price</td>
</tr>
<tr>
<td>Contact Name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning

- Horizontal: Distribute rows into several files
  - Useful for situations where different users need access to different rows
- Vertical: Distribute columns into several relations
  - Useful for situations where different users need access to different columns
  - The primary key must be repeated in each file
- Combined Horizontal and Vertical
- Useful when working with large tables
- Enable parallel processing

Partitions often correspond with User Schemas (user views)

Replication

- Storing pieces of data twice
- Improves performance by minimizing contention
- Integrity problem: duplication
- Best for data not often updated
Designing Physical Files

• Technique for physically arranging records of a file on secondary storage

• File Organizations
  – **Sequential** (Fig. 5-7a): the most *efficient* with storage space.
  – **Indexed** (Fig. 5-7b): *quick retrieval*
  – **Hashed** (Fig. 5-7c): *easiest to update*

Sequential Files

• **Simplest** form of file organization

• New records are **placed** in the **last page** of the file, or if the record does not fit a new page is added to the file.

• If **no order** with respect to values, **linear search** has to be performed to find a specific record.

• If **sorted** by some values, **binary search** can be performed on that value, BUT update/delete/insert operations add overhead
Figure 5-7a Sequential file organization

Records of the file are stored in sequence by the primary key field values

Start of file

Scan

If sorted – every insert or delete requires resort

If not sorted
Average time to find desired record = n/2

Rd f t h f i l

If sorted
every
insert or delete
requires resort

Exam
ple Se
quential File

- For simplicity assume one record per page

<table>
<thead>
<tr>
<th>PAGE</th>
<th>Staff File (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A1 Salzburg 50h</td>
</tr>
<tr>
<td>2</td>
<td>R2 Wien 40h</td>
</tr>
<tr>
<td>3</td>
<td>B3 Salzburg 30h</td>
</tr>
<tr>
<td>4</td>
<td>C1 Salzburg 30h</td>
</tr>
<tr>
<td>5</td>
<td>C2 Salzburg 20h</td>
</tr>
<tr>
<td>6</td>
<td>D2 Wien 10h</td>
</tr>
</tbody>
</table>

Search C1 record (lexicographical order):
1. Retrieve the mid-page of the file Page 3
2. Check ordering field -> B3 is smaller C1 -> search the lower half of the file as new search area and go to page 5
3. C2 is greater C1 hence go to page 4.
Indexes

• Secondary data structures that improve searches

• Primary Index
  – sequentially ordered data file, with an ordering key, used as index

• Clustering Index
  – again sequentially ordered but on a non key field

• Secondary Index
  – index that is defined on a non-ordering field of the data file

Example Indexes

```
<table>
<thead>
<tr>
<th>Primary index</th>
<th>Table</th>
<th>Secondary index</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>Amsterdam</td>
</tr>
<tr>
<td>101</td>
<td>101</td>
<td>London</td>
</tr>
<tr>
<td>102</td>
<td>102</td>
<td>Tokyo</td>
</tr>
<tr>
<td>103</td>
<td>103</td>
<td>New York</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Example Clustering Index

Only one cluster index is possible
(of course, since the clustering index enforces an order on the records)

Example Multilevel Index

- When an index file becomes large the search time within the index file may increase
- A multilevel index tries to reduce the search range by treating the index file as a data file and implementing a sparse index for this file
- The result is a search tree
Search Trees

- Keys and pointers to records are organized in a tree
- In a binary tree, nodes to the left have lower keys
- Unbalanced trees do not help

B(alanced)-Trees

Check animation at:  http://slady.net/java/bt/view.php
Indexing – Performance Analysis

- Sample database schema:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Datatype</th>
<th>Size on Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID (Primary key)</td>
<td>INT</td>
<td>4 bytes</td>
</tr>
<tr>
<td>firstName</td>
<td>CHAR(50)</td>
<td>50 bytes</td>
</tr>
<tr>
<td>lastName</td>
<td>CHAR(50)</td>
<td>50 bytes</td>
</tr>
<tr>
<td>emailAddress</td>
<td>CHAR(100)</td>
<td>100 bytes</td>
</tr>
</tbody>
</table>

- Assume 5 million rows, page size = 1024 bytes
- Blocking factor = ___________
- Number of pages for the table = ______________
- Linear search on ID requires _________ disk accesses
- Binary search on sorted ID requires _________ disk accesses
Indexing – Performance Analysis

• firstName is not sorted, so a binary search is impossible
• Schema for an index on firstName

<table>
<thead>
<tr>
<th>Field_Name</th>
<th>Datatype</th>
<th>Size on Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstName</td>
<td>CHAR(50)</td>
<td>50 bytes</td>
</tr>
<tr>
<td>recordPointer</td>
<td>special</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

• Assume 5 million rows, page size = 1024 bytes
• Blocking factor = _____________
• Number of pages for the index = _______________
• Binary search on index requires ________ disk accesses
  (Plus one more disk access to read the actual record)

Summary of Indexes

• Balanced trees make searching way faster
• Logarithmic instead of linear
  – 10 instead of 1,000
  – 20 instead of 1,000,000
• The base (of the logarithm) is not that important
• Two main kinds
  – Search trees (or B-trees)
  – Hash tables (next)
Hash Tables

- Place records (or key and pointer) in buckets
- Use a function on the key to find the appropriate bucket
- Problems
  - Collisions (several keys map to the same bucket)
  - Overflow (too many keys for the bucket)
- Most good functions help only with point queries

Figure 5-7c

Hashed file or index organization

Animation @ http://www.cs.pitt.edu/~kirk/cs1501/animations/Hashing.html
Example Hash File

- Let \( x \) be the digits in StaffNumber
- Hash Function \( H(x) = x \mod 3 \)

Now insert B21 Wien:
\[ H(21) = 21 \mod 3 = 0 \]

What do we do when we want to insert L41 Graz? (bucket 2)

Resolving Collisions

- Open Addressing
  - Search the file for the first available slot to insert the record
- Overflow Area
  - Instead of searching a free slot, maintain an overflow area collisions
- Multiple Hashing
  - Applies a second hash function if the first one results in a collision. Usually used to place records in an overflow area.
## Example Open Addressing

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Staff File (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A9 Salzburg</td>
</tr>
<tr>
<td></td>
<td>B21 Wien</td>
</tr>
<tr>
<td>1</td>
<td>B37 Salzburg</td>
</tr>
<tr>
<td>2</td>
<td>D5 Salzburg</td>
</tr>
<tr>
<td></td>
<td>S14 Wien</td>
</tr>
</tbody>
</table>

Insert L41 Graz

## Example Overflow Area

**Pointer to Overflow Area**

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Staff File (data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A9 Salzburg</td>
</tr>
<tr>
<td></td>
<td>B21 Wien</td>
</tr>
<tr>
<td>1</td>
<td>B37 Salzburg</td>
</tr>
<tr>
<td>2</td>
<td>D5 Salzburg</td>
</tr>
<tr>
<td></td>
<td>S14 Wien</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Overflow area</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>L41 Graz</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
File Organizations -- Summary

• Unordered
• Sequential / Sorted
  – Problem: reorganization
• Primary Indexed
  – Records are organized according to an index
  – May help with sequential scan (sorted)
• Secondary Indexed
  – Create index for some key (not affecting records)
• Clustered
  – Store several kinds of records on the same page

Kinds of Operations (low-level)

• Point query
  – Find all records with a specific value on a field
• Range query
  – Find all records with a field value in a given range
• Full scan
  – Go through all rows in a table
• Sequential scan
  – Go through all rows in a table in a certain order
• Joins
Query Optimization

• SQL is declarative
  – doesn’t say HOW to do the query

• DBMS has choices
  – Reordering operations
  – Using indexes or not
  – Joins: starting from which side
  – Different algorithms

• Modern DBMS use Cost-based optimizers
  – Generate many possibilities
  – Estimate the ‘cost’ of each possibility
  – Do the ‘cheapest’ one

SQL Indexes

• SQL indexes can be created on the basis of any selected attributes

  CREATE INDEX student_name_idx
  ON Student (Name);

  CREATE CLUSTERED INDEX student_age_idx
  ON Student (Age);
SQL Indexes (contd.)

You may even create an index that prevents you from using a value that has been used before. Such a feature is especially useful when the index field (attribute) is a primary key whose values must not be duplicated:

```
CREATE UNIQUE INDEX <index_field>
ON <tablename> (the key field);

DROP INDEX <index_name> ON <tablename>;
```

Rules for Using Indexes

1. Use on larger tables
2. Index the primary key of each table
3. Index search fields (fields frequently in WHERE clause)
4. Fields in SQL ORDER BY and GROUP BY commands
5. When there are >100 values but not when there are <30 values
Rules for Using Indexes (cont.)

6. Avoid use of indexes for fields with long values; perhaps compress values first

7. If key to index is used to determine location of record, use surrogate (like sequence nbr) to allow even spread in storage area

8. DBMS may have limit on number of indexes per table and number of bytes per indexed field(s)

9. Be careful of indexing attributes with null values; many DBMSs will not recognize null values in an index search