INFO2120 – INFO2820 – COMP5138 Database Systems

Week 10: Indexing and Tuning

(Kifer/Bernstein/Lewis – Chapter 9.4 & 12; Ramakrishnan/Gehrke – Chapter 8; Ullman/Widom – Chapter 8.3)

Dr. Uwe Röhm
School of Information Technologies

This Week's Learning Objectives

- **Database Storage Layer: Physical Data Organisation**
  - Motivation: Data stored on disks
  - Several physical design alternatives possible for same logical schema
- **Indexing of Databases**
  - Efficient data access based on *search keys*
  - Several design decisions…
- **Database Tuning**
  - How to suggest appropriate indexes for a given SQL workload
  - Awareness of the trade-off between query performance and indexing costs (updates)

Based on slides from Kifer/Bernstein/Lewis (2006) "Database Systems" and from Ramakrishnan/Gehrke (2003) "Database Management Systems", and including material from Fekete and Roehm

Motivation: Disk Storage

- DBMS stores information on ("hard") disks.
  - *Main memory is much more expensive than HDDs.*
    A$100 will buy you either 16 GB of RAM or 2 TB of disk today.
  - *Main memory is volatile.*
    We want data to be saved between runs. (Obviously!)
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
  - Indeed, overall performance is determined largely by the number of disk I/Os done

Storage Hierarchy

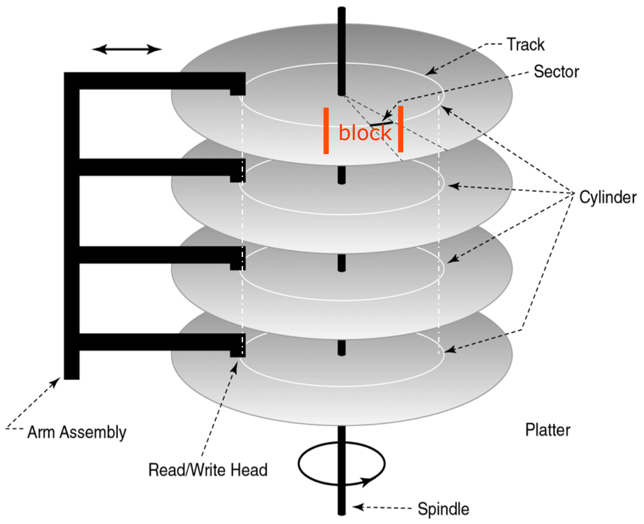
- **primary storage:** Fastest media but volatile (cache, RAM).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g.: hard disks, solid-state drives
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

Disks

- Currently the secondary storage device of choice.
- Capable of storing large quantities of data cheaply
  - Main advantage over tapes: random access vs. sequential.
- Non-volatile
- Extremely slow compared with CPU speed
  - **Access Gap** between primary and secondary storage of order of magnitudes
  - **Trends: Disk capacity is growing rapidly, but access speed is not!**
- Data is stored & retrieved in units called disk blocks or pages.
- Performance of DBMS largely a function of the number of disk I/O operations that must be performed

Physical Disk Structure

- The platters spin (say, 7200rpm).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a cylinder (imaginary).
- Only one head reads/writes at any one time.
- Block size is a multiple of sector size (which is fixed).
  - typically sizes: 4kB - 8kB



Physical organization of a disk storage unit.

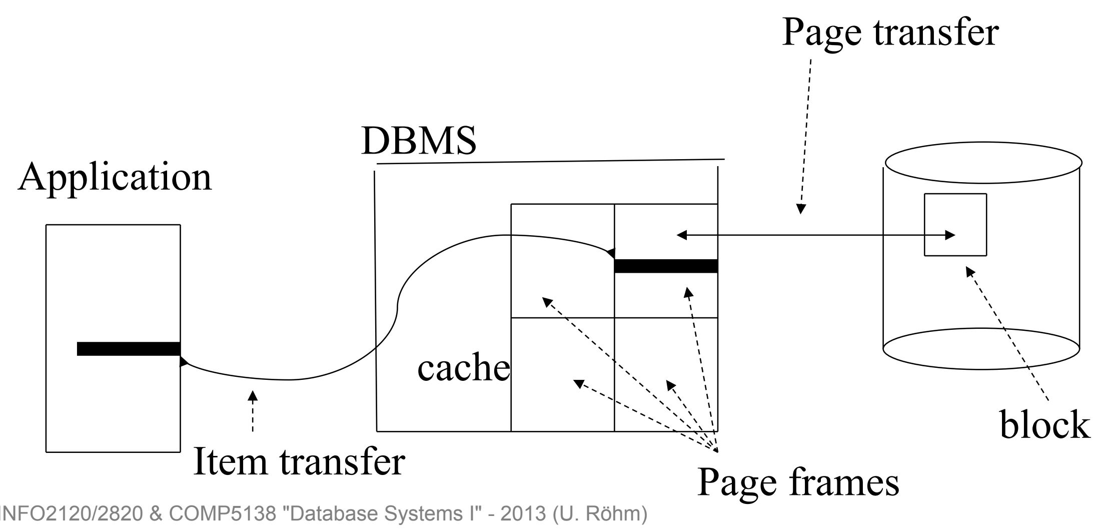
[Kifer/Bernstein/Lewis, 2006]

Accessing a Database Disk Page

- **Time to access (read/write) a disk block:**
  - seek time (moving arms to position disk head on track)
  - rotational delay (waiting for block to rotate under head)
  - transfer time (actually moving data to/from disk surface)
- **Seek time and rotational delay dominate.**
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- **Key to lower I/O cost: reduce seek/rotation delays!**

Storage Access

- A database file is partitioned into fixed-length storage units called data page). Pages are units of both storage allocation and data transfer.
- Idea 1: Store related information close together on disk ('clustered')
- Idea 2: *Database Buffer*: Database system seeks to minimize the number of block transfers between the disk and memory.



Organization of Records in Files

- The database is stored as a **collection of files**. Each file is a sequence of records. A **record** is a sequence of fields.
  - Typically: **Tablespace** as logical unit to hold *n* tables in different files

- OS does disk space & buffer management: why not leave OS to manage these tasks for the DBMS?
  - Differences in OS support: portability issues
  - Some limitations, e.g., files can't span disks.
  - Buffer management in DBMS requires ability to pin pages etc.

This Week's Agenda

- Overview of the Database Storage Layer
- **Access Paths and Indexing**
- Choosing an Index

How to Store a Database?

- Logical Database Level:
  - A database is a collection of ***relations***. Each relation is a set of ***records*** (or *tuples*). *A* record is a sequence of ***fields*** *(or attributes*).
  - Example:

```
CREATE TABLE Student (
   id        INTEGER     PRIMARY KEY,
   name      VARCHAR(40) UNIQUE,
   address   VARCHAR(255),
   gender    CHAR(1),
   birthdate DATE
);
```

- Physical Database Level:
  - How to represent tuples with several attributes (*fields*)?
  - How to represent collection of tuples and whole tables?
  - How do we find specific tuples?

Access Path

- Refers to the *algorithm* + *data structure* (e.g., an index) used for retrieving and storing data in a table
- The choice of an access path to use in the execution of an SQL statement has no effect on the semantics of the statement
- This choice can have a major effect on the execution time of the SQL statement

=> Physical Data Independence

Alternative File Organizations

Many alternatives exist, each ideal for some situations, and not so good in others:

- **Heap Files** – a record can be placed anywhere in the file where there is space (random order)
  - suitable when typical access is a *file scan* retrieving all records.
- **Sorted Files** – store records in sequential order, based on the value of the search key of each record
  - best if records must be retrieved in some order, or only a `*range*' of records is needed.
- **Indexes** – data structures to organize records via trees or hashing
  - like sorted files, they speed up *searches for a subset of records*, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

(Unordered) Heap Files

- Simplest file structure contains records in no particular order.
- Access method is a ***linear scan***
  - In average half of the pages in a file must be read, in the worst case even the whole file
  - Efficient if all rows are returned (SELECT * FROM *table*)
  - Very inefficient if a *few* rows are requested
- Rows appended to end of file as they are inserted
  - Hence the file is unordered
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

Example: Transcript Stored as Heap File

666666	MGT123	F1994	4.0	page 0
123456	CS305	S1996	4.0	
987654	CS305	F1995	2.0	

717171	CS315	S1997	4.0	page 1
666666	EE101	S1998	3.0	
765432	MAT123	S1996	2.0	
515151	EE101	F1995	3.0	

234567	CS305	S1999	4.0	page 2
878787	MGT123	S1996	3.0	

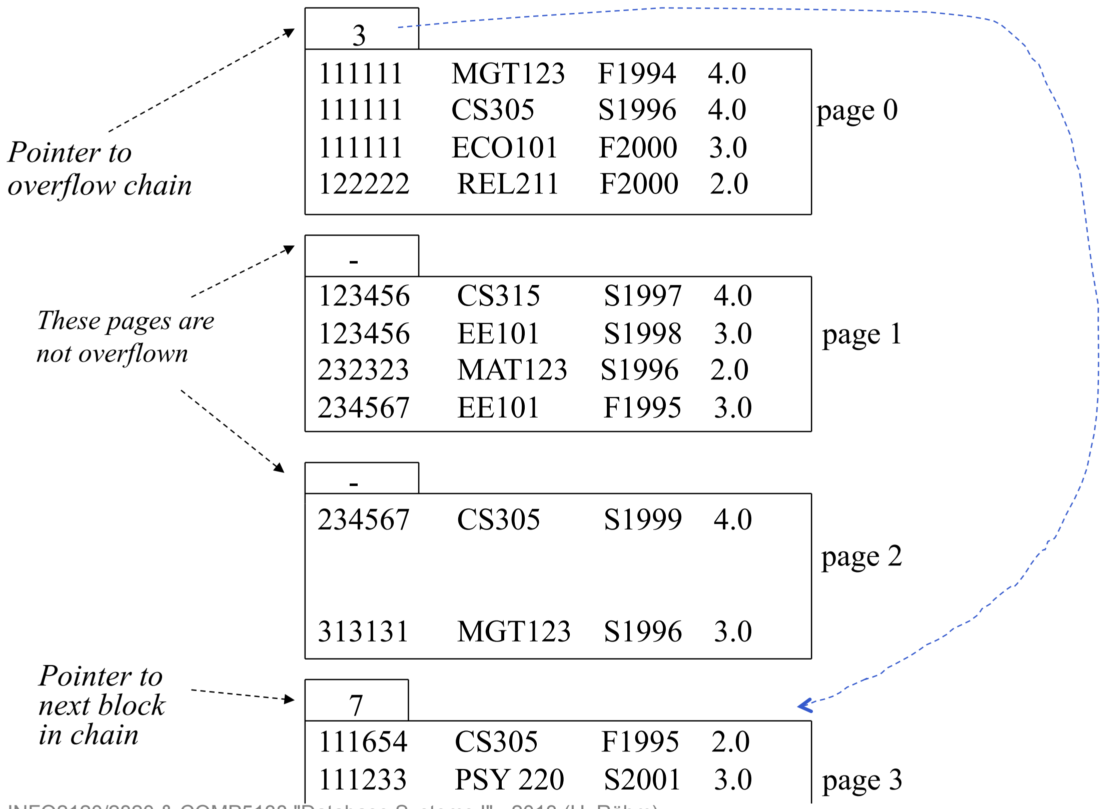
Sorted File

- Rows are sorted based on some attribute(s)
  - Successive rows are stored in same (or successive) pages
- Access method could be a ***binary search***
  - Equality or range query based on that attribute has cost log_2B to retrieve page containing first row
- Problem: Maintaining sorted order
  - After the correct position for an insert has been determined, shifting of subsequent tuples necessary to make space (very expensive)
  - Hence sorted files typically are not used per-se by DBMS, but rather in form of index-organised (clustered) files

Example: Transcript as Sorted File

111111	MGT123	F1994	4.0	page 0
111111	CS305	S1996	4.0	
123456	CS305	F1995	2.0	
123456	CS315	S1997	4.0	page 1
123456	EE101	S1998	3.0	
232323	MAT123	S1996	2.0	
234567	EE101	F1995	3.0	
234567	CS305	S1999	4.0	page 2
313131	MGT123	S1996	3.0	

Overflow Example (after Inserts)


Pointer to overflow chain
These pages are not overflown
Pointer to next block in chain
3
-
-
7
page 0
page 1
page 2
page 3

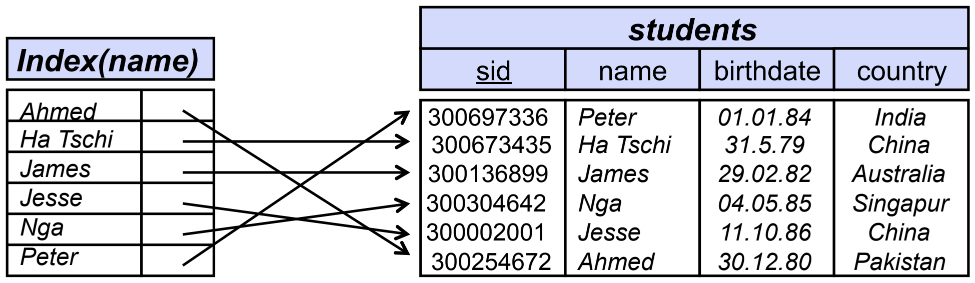
3	111111	MGT123	F1994	4.0	page 0
	111111	CS305	S1996	4.0	
	111111	ECO101	F2000	3.0	
	122222	REL211	F2000	2.0	
-	123456	CS315	S1997	4.0	page 1
	123456	EE101	S1998	3.0	
	232323	MAT123	S1996	2.0	
	234567	EE101	F1995	3.0	
-	234567	CS305	S1999	4.0	page 2
	313131	MGT123	S1996	3.0	
7	111654	CS305	F1995	2.0	page 3
	111233	PSY 220	S2001	3.0	

Indices

- Can we come up with a file organisation that is
  - as efficient for searches (especially on ranges) as an ordered file?
  - as flexible as a heap file for inserts and updates?

- Idea: Separate location mechanism from data storage
  - Just remember a book index:
    Index is a set of pages (a separate file) with pointers (page numbers) to the data page which contains the value
  - Instead of scanning through whole book (relation) each time, using the index is much faster to navigate (less data to search)
  - Index typically much smaller than the actual data

Index Example



Index(name)

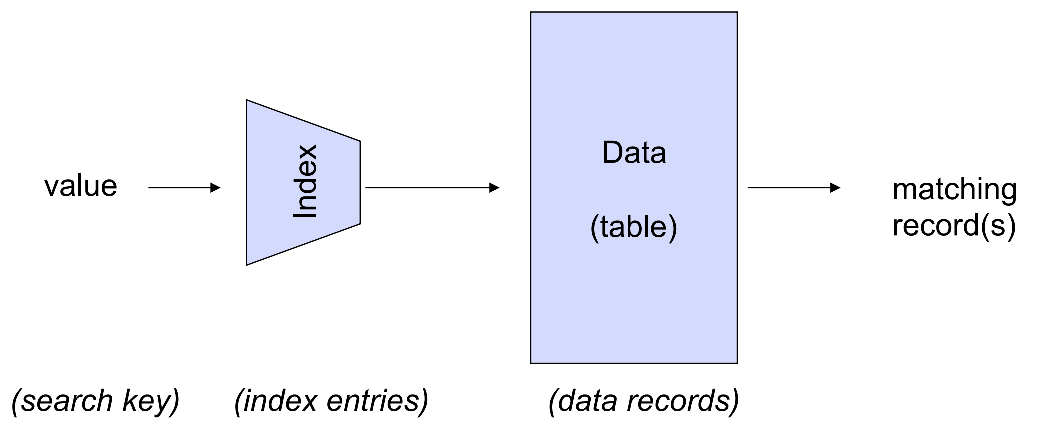
Ahmed	
Ha Tschi	
James	
Jesse	
Nga	
Peter	

students

sid	name	birthdate	country
300697336	*Peter*	*01.01.84*	*India*
300673435	*Ha Tschi*	*31.5.79*	*China*
300136899	*James*	*29.02.82*	*Australia*
300304642	*Nga*	*04.05.85*	*Singapur*
300002001	*Jesse*	*11.10.86*	*China*
300254672	*Ahmed*	*30.12.80*	*Pakistan*

- **Ordered index**: search keys are stored in sorted order
- **Hash index**: search keys are distributed uniformly across "buckets" using a "hash function".

The General Picture



Note: ‘Cow book’ (Ramakrishnan/Gehrke) means with index entries only the location mechanism, while it calls the final index entries ‘data entries’; we’ll stick with the notation in the ‘Blue Book’

Indices

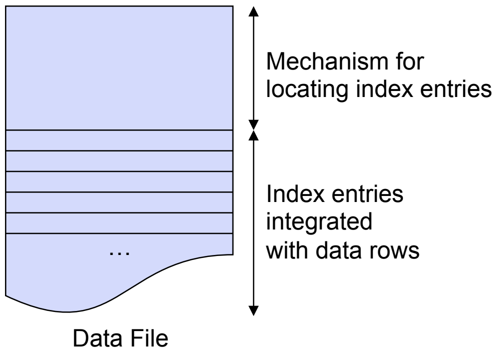
- An **index** is an *access path* to efficiently locate row(s) via *search key fields* without having to scan the entire table.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as key (minimal set of fields that uniquely identify a record in a relation).
    - Candidate key: *set* of attributes; *guarantees* uniqueness
    - Search key: *sequence* of attributes; *does not guarantee* uniqueness – just used for search
  - Primary keys are typically automatically indexed

- Index consists of
  - *Location mechanism* (directing search)
  - *Index entries* (either containing data records or pointers to them)
  - If index entries contain actual records (rows),called **integrated index**

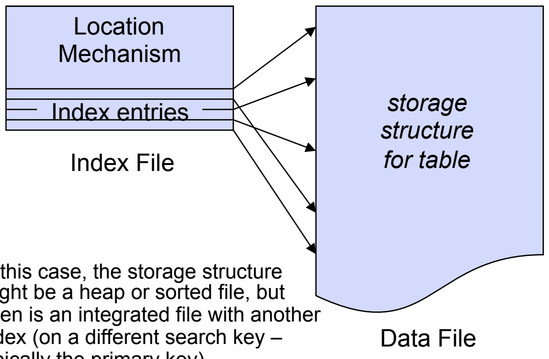
Storage Structure

- Structure of file containing a table
  - Heap file (no index, not integrated)
  - Sorted file (no index, not integrated)
  - Integrated file containing index and rows (index entries contain rows)
    - B^+ tree
    - Hash

Integrated Storage Structure
contains table and (primary) index:


Mechanism for locating index entries
Index entries integrated with data rows
…
Data File

Non-integrated Storage Structure
index and storage structure are separated:



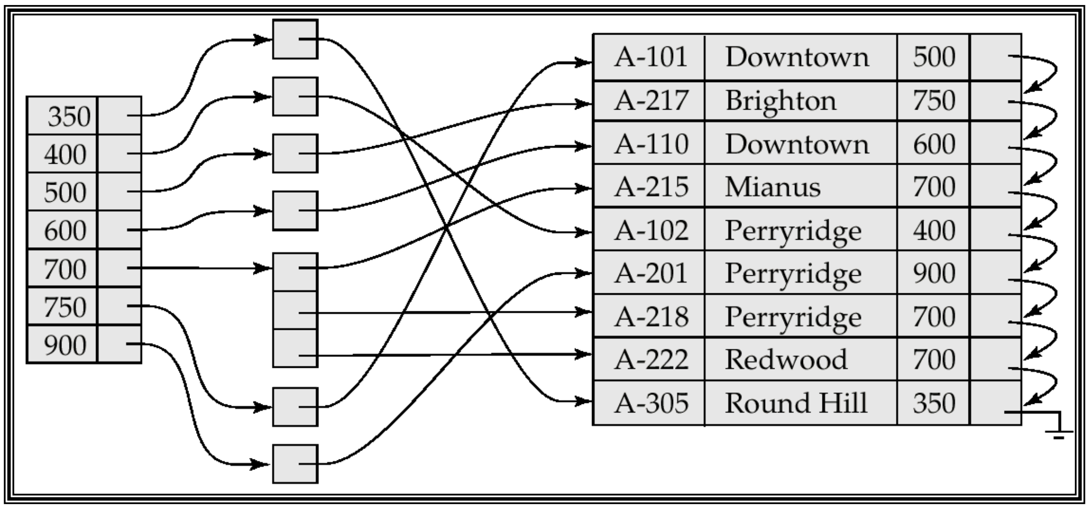
In this case, the storage structure might be a heap or sorted file, but often is an integrated file with another index (on a different search key – typically the primary key)

Primary vs. Secondary Indices

- **Primary index**:
  index whose search key specifies the sequential order of file
  - Also called **main index** or **integrated index**
  - Note: Most literature does not distinguish between main and primary index and just refers to a main index also as the primary index…
  - Exception of this is PostgreSQL: it calls an index a primary index if its search key contains the primary key regardless of data order…
- **Secondary index**: an index whose structure is separated from the data file and whose search key typically specifies an order different from the sequential order of the file.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Because each record access may fetch a new block from disk
  - At least as long as the secondary index is not clustered (see next slides)

Example: Secondary Index

- Secondary Index on **balance** field of **account**
- As balance is not a candidate key, we need buckets as an indirection with pointers to tuples with the same search-key



Source: Silberschatz/Korth/Sudarshan: *Database System Concepts*, 2002.

Index Definition in SQL

- Create an index
  CREATE INDEX *name* ON *relation-name* (<*attributelist*>)
  - Example:
    CREATE INDEX *StudentName* ON Student(*name*)
- Index on primary key generally created automatically
  - Use **CREATE UNIQUE INDEX** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index
  DROP INDEX *index-name*
- Sidenote: SQL-92 does actually not officially define commands for creation or deletion of indices.
  - vendors kind-of 'agreed' to use this syntax consistently

Indices - The Downside

- Additional I/O to access index pages (except if index is small enough to fit in main memory)
  - The hope is that this is less than the saving through more efficient finding of data records
- Index must be updated when table is modified.
  - depends on index structure, but in general can become quite costly
  - so every additional index makes update slower…
- Decisions, decisions…
  - Index on primary key is generally created automatically
  - Other indices must be defined by DBA or user, through vendor specific statements
  - Choose which indices are worthwhile, based on workload of queries (cf. later this lecture)

Index Classification

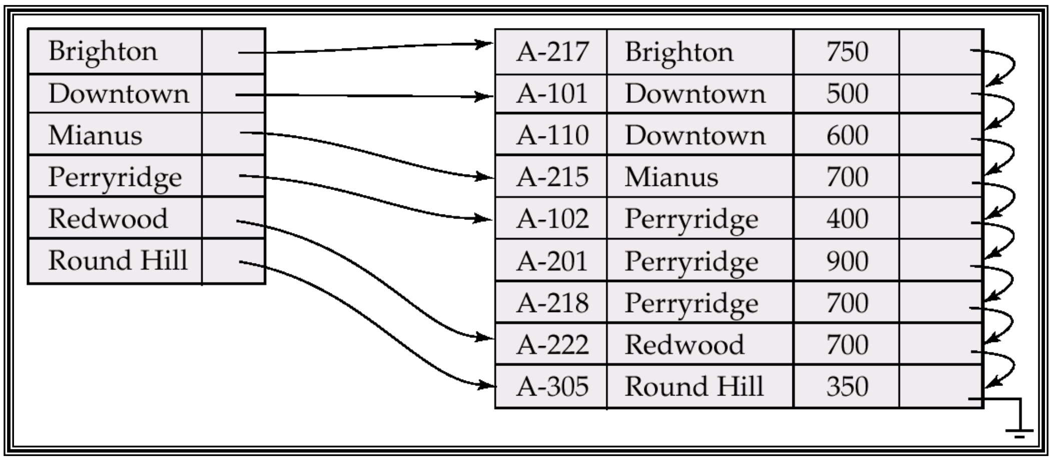
- Primary Index vs. Secondary
  - If index entries contain actual data rows, then called **main index**.
    - Oracle calls this *integrated storage structure* an 'index-organised table'
    - Note: Some literature refers to this as primary index
  - Otherwise **secondary index**
- Unique vs. Non-Unique
  - an index over a candidate key is called a **unique index** (no duplicates)
- Single-Attribute vs. Multi-Attribute
  - whether the search key has one or multiple fields
- Clustered vs. Unclustered
  - If data records and index entries are ordered the same way, then called **clustering index**.

Clustering Index

- ***Clustering index***:
  index entries and rows are ordered in the same way
  - The particular index structure (eg, hash, tree) dictates how the rows are organized in the storage structure
    - There can be at most one clustering index on a table
  - An integrated storage structure is always clustered (since rows and index entries are the same)
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key

Example: Clustered Index

- Clustered Index on **branch-name** field of **account**

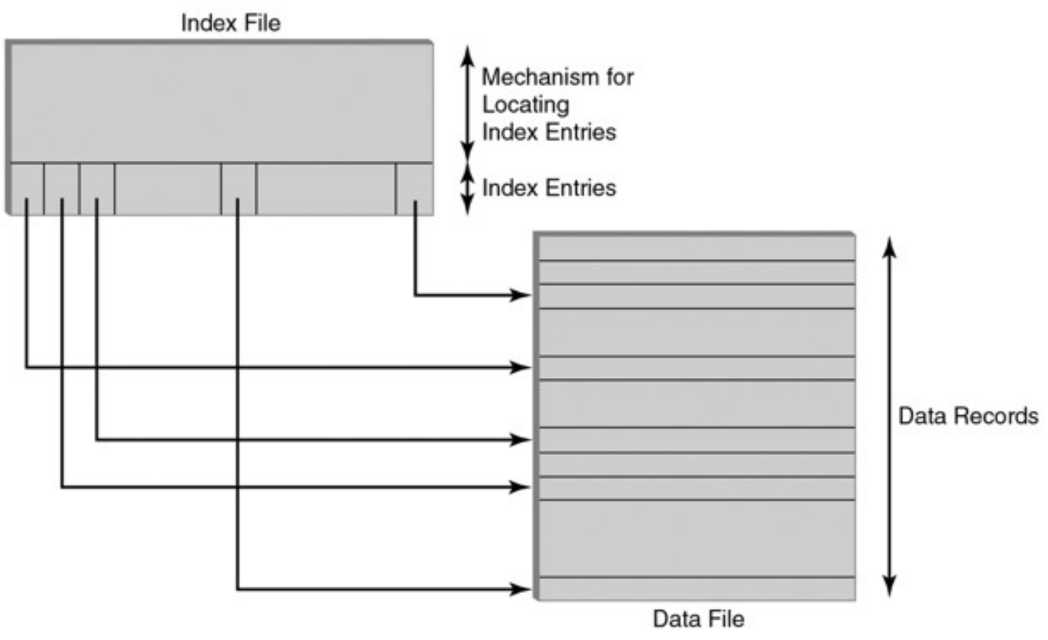


Source: Silberschatz/Korth/Sudarshan: *Database System Concepts*, 2002.

Unclustered Index

- Unclustered (secondary) index:
  index entries and rows are not ordered in the same way
- An secondary index might be clustered or unclustered with respect to the storage structure it references
  - It is generally unclustered (since the organization of rows in the storage structure depends on main index)
  - There can be many secondary indices on a table
  - Index created by CREATE INDEX is generally an unclustered, secondary index

Unclustered Secondary Index



Clustered vs. Unclustered Indices

- Clustered Index: Good for range searches over a range of search key values
  - Use index to locate first index entry at start of range
    - This locates first row.
  - Subsequent rows are stored in successive locations if index is clustered (not so if unclustered)
  - Minimizes page transfers and maximizes likelihood of cache hits
- Example: Access Costs of a Range Scan
  - Data file has 10,000 pages, 100 rows in search range
  - Page transfers for table rows (assume 20 rows/page):
    - Heap: 10,000 (entire file must be scanned)
    - File sorted on search key: log_2 10000 + (5 or 6) ≈ 19
    - Unclustered index: ≤ *index-height* + 100
    - Clustered index: *index-height* + (5 or 6)

Comparison

- ***Clustered index***: index entries and rows are ordered in the same way
- There can be at most one clustered index on a table
  - CREATE TABLE generally creates an integrated, clustered (main) index on primary key
- Especially good for "range searches" (where search key is between two limits)
  - Use index to get to the first data row within the search range.
  - Subsequent matching data rows are stored in adjacent locations (many on each block)
  - This minimizes page transfers and maximizes likelihood of cache hits

- ***Unclustered (secondary) index***: index entries and rows are not ordered in the same way
- There can be many unclustered indices on a table
  - As well as perhaps one clustered index
  - Index created by CREATE INDEX is generally an unclustered, secondary index
- Unclustered isn't ever as good as clustered, but may be necessary for attributes other than the primary key

Indexing in the "Physical World"

- Library:

  ```
  CREATE TABLE Library (
      callno  CHAR(20)  PRIMARY KEY,
      title      VARCHAR(255),
      author  VARCHAR(255),
      subject VARCHAR (128)
   )
  ```

  - books are ordered by call number (callno).
  - However, we typically search by title, author, subject/keyword

Indexing in the "Physical World"

- Library stacks are "clustered" by call number.
- The catalog is a **secondary** index…say by Title
- **CREATE Index TitleCatalog on Library(title)**
- Note secondary index is always *dense* – an index entry for each record
  - In contrast to the *sparse* indexes of the call number ranges that are displayed at the head-side of each library stack

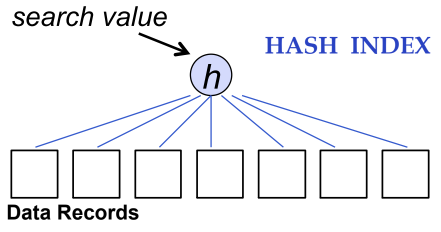
Multicolumn Search Keys

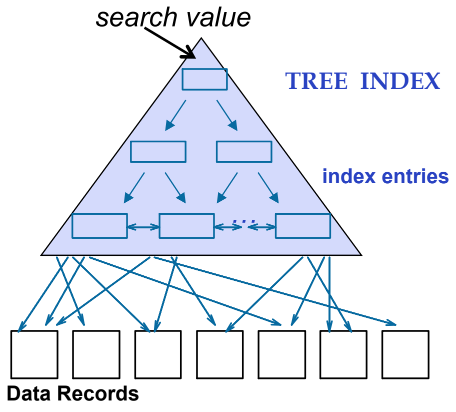
- CREATE INDEX Inx ON Tbl (*Att1*, *Att2*)
- Search key is a *sequence* of attributes; index entries are lexically ordered
- Supports finer granularity equality search:
  - "Find row with value (A1, A2) "
- Supports range search (tree index only):
  - "Find rows with values between (A1, A2) and (A1′, A2′) "
- Supports partial key searches (tree index only):
  - Find rows with values of *Att1* between A1 and A1′
  - But not "Find rows with values of *Att2* between A2 and A2′ "
- Supports that an index <u>covers</u> a whole query
  - Index on (Y,X) can answer "**`SELECT X FROM table WHERE Y=`*`const`***"

Which Types of Indexes are available?

- Tree-based Indexes: B+-Tree
  - *Very flexible, only indexes to support point queries, range queries and prefix searches*
- Hash-based Indexes
  - *Fast for equality searches - and that's it*
- Special Indexes
  - Such as Bitmap Indexes for OLAP or R-Tree for spatial databases

Found in every database engine





=> More details on disk-based index structures in INFO3404

Multi-Level Tree Index

Most DBMS use B+-tree data structure

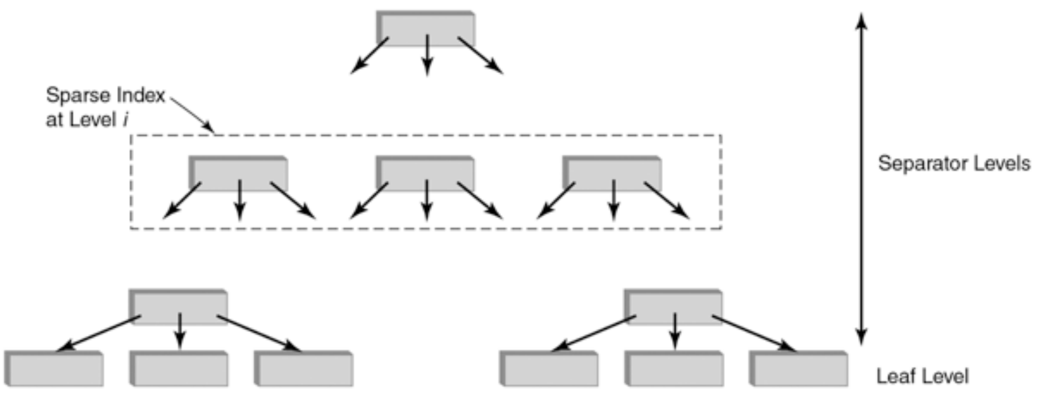


FIGURE 9.14 Schematic view of a multilevel index.

- Locate data records by descending the tree from root to leaf
  - Search cost to find pointer to data row(s) = number of levels in index tree
  - This is logarithmic in theory, but in practice can be considered as a small constant!
  - Typical index with 2 or 3 levels can support millions of data rows

DBMS Comparison: Index Types

	DB2 UDB 8.2	Oracle 10g	SQLServer 2008	Sybase ASE 12.5	Postgres 9	MySQL 5
B+-Tree	yes	yes	yes	yes	yes	yes
Hash Index	---	yes	---	---	yes	MEMORY tables
Bitmap Index	(yes) (called EVI)	yes (since v8.1)	*Bitmap filter* ---	yes (in Adaptive IQ)	*bitmap scan* (since v8.1)	---
Specialities	*R-Tree(*)*	*R-Tree (*)*	Quad Tree; fulltext index	---	Inverted idx; GiST	Fulltext *R-Tree*
Integrated (Main) Index	no?	yes	yes	yes	---	InnoDB (always PK)
Clustered Index	yes	yes (only as so-called index-organised table)	yes (every clustered index is an integrated index)	yes (a clustered index is an integrated index)	yes	InnoDB (always PK)
Unique Index	yes	yes	yes	yes	yes	yes
Multi-Column Index	yes	yes	yes	yes	yes	yes

(*) spatial index via extension module

Database Metadata

- Data dictionary (also called system catalog) stores metadata such as:
- Information about relations
  - names of relations
  - names and types of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation
    - (operating system file names or disk addresses etc)
- Information about indices
- Typically stored as a set of relations (e.g. Oracle: USER_TABLES etc.)

INFORMATION_SCHEMA

- SQL-92 standard for database meta-data: INFORMATION_SCHEMA
  - Supported by many DBMS such as SQL Server, MySQL, PostgreSQL, (but not Oracle)
- `INFORMATION_SCHEMA.*`
  - `.Schemata`
  - `.Tables`
  - `.Views`
  - `.Columns`
  - `.Statistics`
  - `.Table_Constraints`
  - `.Referential_Constraints`
  - `.Check_Constraints`
  - `.Triggers`
  - `.User_Privileges`
  - …

Note:
`INFORMATION_SCHEMA.Indexes` not part of the standard (because CREATE INDEX is not in SQL-92)

cf: http://www.postgresql.org/docs/9.2/interactive/information-schema.html
http://dev.mysql.com/doc/refman/5.0/en/information-schema.html
http://msdn.microsoft.com/en-us/library/ms186778.aspx

This Week's Agenda

- Overview of the Database Storage Layer
- Indexing
- **Choosing an Index**

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choices of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash or Tree?
- How should the indexes be created?
  - Separate tablespace? Own disk?
  - Fillfactor for index nodes?

Choices of Indexes (cont'd)

- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query only supported by tree index types.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable **index-only** strategies for important queries (so-called *covering index)*.
    - For index-only strategies, clustering is not important!
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.
- Create indexes in own tablespace on separate disks

Choosing an Index

- An index should support a query of the application that has a significant impact on performance
  - Choice based on frequency of invocation, execution time, acquired locks, table size
- Example 1:

```
SELECT E.Id
  FROM Employee E
 WHERE E.Salary < :upper  AND  E.Salary > :lower
```

  - This is a range search on *Salary*.
  - Since the primary key is *Id*, it is likely that there is a clustered, main index on that attribute that is of no use for this query.
  - Choose a secondary, B+ tree index with search key *Salary*

Covering Index

- Goal: Is it possible to answer whole query just from an index?
- **Covering Index** - an index that contains all attributes required to answer a given SQL query:
  - all attributes from the WHERE filter condition
  - if it is a grouping query, also all attributes from GROUP BY & HAVING
  - all attributes mentioned in the SELECT clause
- Typically a multi-attribute index (*Q: why 'typically'?*)
  - Order of attributes is important: Prefix of the search key must be the attributes from the WHERE (*Q: why?*)
- Example:
  - For query on previous slide it is (,)

Choosing an Index (cont'd)

- Example 2:

```
SELECT T.studId
  FROM Transcript T
 WHERE T.grade = :grade
```

- This is an equality search on *grade*.
- We know the primary key is *(studId, semester, uosCode)*
  - It is likely that there is a main, clustered index on these PK attributes
  - but it is of no use for this query…
- Hence: Choose a secondary, B+ tree index (or hash index) with search key *Grade*
  - Again: a *covering* index with composite search key *(grade, studId)* would allow to answer complete query out of index
    - but then only as B-Tree index…

Choosing an Index (cont' d)

- Example 3:

```
SELECT T.uosCode, T.grade
  FROM Transcript T
 WHERE T.studId = :id  AND  T.semester = 'Sem1'
```

- This is an equality search on *studId* and *semester*.
- If the primary key is *(studId, semester, uosCode)*, it is likely that there is a main, clustered index on these sequence of attributes
  - If the main index is a B+ tree, it can be used for this search (requested attributes are a prefix of the search key)
  - If the main index is a hash index, it cannot be used for this query. Then choose a secondary, B+ tree or hash index either with search key *studId* (since *semester* is not as selective as *studId*) or with a composite search key on *(studId, semester)*
- Suppose the primary key is *(uosCode, studId, semester*).
  - Then the main index is of no use (independent of whether it is a hash or a B+ tree).

Choosing an Index (cont' d)

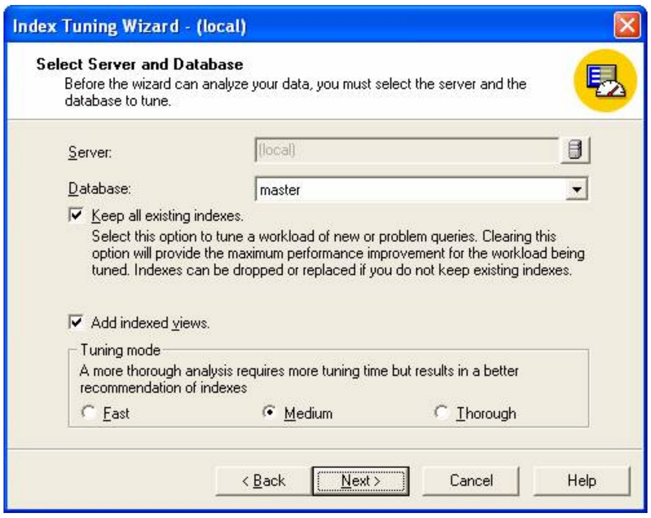
- Example 4:

```
SELECT T.uosCode, COUNT(*)
  FROM Transcript T
 WHERE T.year = 2009 AND T.semester = 'Sem1'
 GROUP BY T.uosCode
```

- This is a group-by query with an equality search on *year* and *semester*.
- If the primary key is *(studId, year, semester, uosCode)*, it is likely that there is a primary (clustered) index on these sequence of attributes
  - But the search condition is on *year* and *semester* => must be prefix!
  - Hence PK index not of use
  - Covering INDEX: *(year, semester, uosCode)*

Index Tuning Wizards

- Many modern DBMS have a tuning advisor component
  - SQL Server
  - IBM DB2
  - Oracle
  - …



Index Tuning Wizard - (local)

Select Server and Database

Before the wizard can analyze your data, you must select the server and the database to tune.

Server: (local)

Database: master

[✓] Keep all existing indexes.

Select this option to tune a workload of new or problem queries. Clearing this option will provide the maximum performance improvement for the workload being tuned. Indexes can be dropped or replaced if you do not keep existing indexes.

[✓] Add indexed views.

Tuning mode

A more thorough analysis requires more tuning time but results in a better recommendation of indexes

Fast [✓ Medium] Thorough

- Make recommendations on index (Work: materialized views) based on given SQL commands / workload

Lessons Learned this Week

- **Physical Data Organisation**
  - Core Problem: Persistency requires disks, but those are SLOW
  - Several physical design alternatives possible for same logical schema
- **Understanding of the concept of an Index**
  - Efficient access to single tuples or even ranges based on *search keys*
  - One *primary*, but several *secondary indices* possible per relation
  - single- vs. *multi-attribute indices*, *clustered* vs. *unclustered indices*
- **Practical experience with indexing a relational database**
  - How to suggest appropriate indexes for a given SQL workload
  - Awareness of the trade-off between SQL query performance and indexing costs (updates)

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 9 (9.1-9.4)
  - Chapter 12 (database tuning)
  - *Kifer/Bernstein/Lewis gives a good overview of indexing and especially on how to use them for database tuning. This is the focus for INFO2120 too.*
- Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)
  - Chapter 8
  - *The Ramakrishnan/Gehrke is very technical on this topic, providing a lot of insight into how disk-based indexes are implemented. We only need the overview here (Chap8); technical details are covered in info3404.*
- Ullman/Widom (3rd edition - '1st Course in Databases')
  - Chapter 8 (8.3 onwards)
  - *Mostly overview, but cost model of indexing goes further than we discuss here in the lecture*

- [Oracle 10g Database Concepts, Chap. 5.4]
  Oracle Corporation: Oracle 10g Documentation, *Database Concepts.*

Next Week

- OLAP and Data Warehousing
  - Issues and the ETL process
  - Star Schema
  - CUBE and ROLLUP Operators
  - ROLAP / MOLAP

- Readings:
  - Kifer/Bernstein/Lewis book, Chapter 17 (up-to 17.7)
  - <u>or</u> alternatively (if you prefer those books):
    - Ramakrishnan/Gehrke (Cow book), Chapter 25