Intro

Database: Set of data, organized in some fashion for easy retrieval.
Data Model: a formalism to describe "constraints" that describe "properties" of data.
Relational: store data as rows and columns
OO: store data as objects
OO is more powerful that Relational
high expressive power(type or nature; what kind of constrains I can specific natively) is good but we still care about Relational.
OO and Relational are general purpose DBMS: so expressive that can capture many apps. But currently design logical models for certain apps to get optimized performance.
Schema: a set of constraints that:

- describe the "properties" of data
- describe the structure of the data.
- organize the data

Schema is described within the formalism corresponding to the underlying data model.
Classification of models/schemas:
Physical: Data structures, data layout in memory or cache
Logical: Relational(is almost Physical model), OO(is closer to Conceptual Model, easy for programmer), OR
Conceptual: UML, ER, Extended ER
Things in conceptual level will be less predictable, whereas in physical model is the opposite.
DBMS: A software/hardware system for storage, retrieval, manipulation of data.
Why use a DBMS(reducing cost):

- Data independence and efficient access
- Reduced application development time.
- Data integrity and security.
- Uniform data administration
- Concurrent access, recovery from crashes(may have impact on scalability).

Why DBMS for concurrence not OS:
there are many other metrics be provided by DBMS. A DBMS handles reads and writes, transactions on different objects in a DB. While OS usually dealing with files. DBMS can be very efficient and flexible. because small numbers of predictable ways to access to data. Access patterns also is very predictable. OS is difficult to predict.
Data Independence: Applications insulated from how data is structured and stored.

- Logical data independence: Protection from changes in logical structure of data.
- Physical data independence: Protection from changes in physical structure of data.

Levels of Abstraction in a DBMS:
The database description consists of a schema at each of these three levels of abstraction:

- conceptual: The conceptual schema (sometimes called the logical schema) describes the stored data in terms of the data model of the DBMS.
- physical: The physical schema specifies additional storage details. Essentially, the physical

schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes.

- external: allow data access to be customized (and authorized) at the level of individual users or groups of users.

Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users.

DB Generations:

First generation: Hierarchical & network, 60s' nested structure(like a tree) and object & reference.

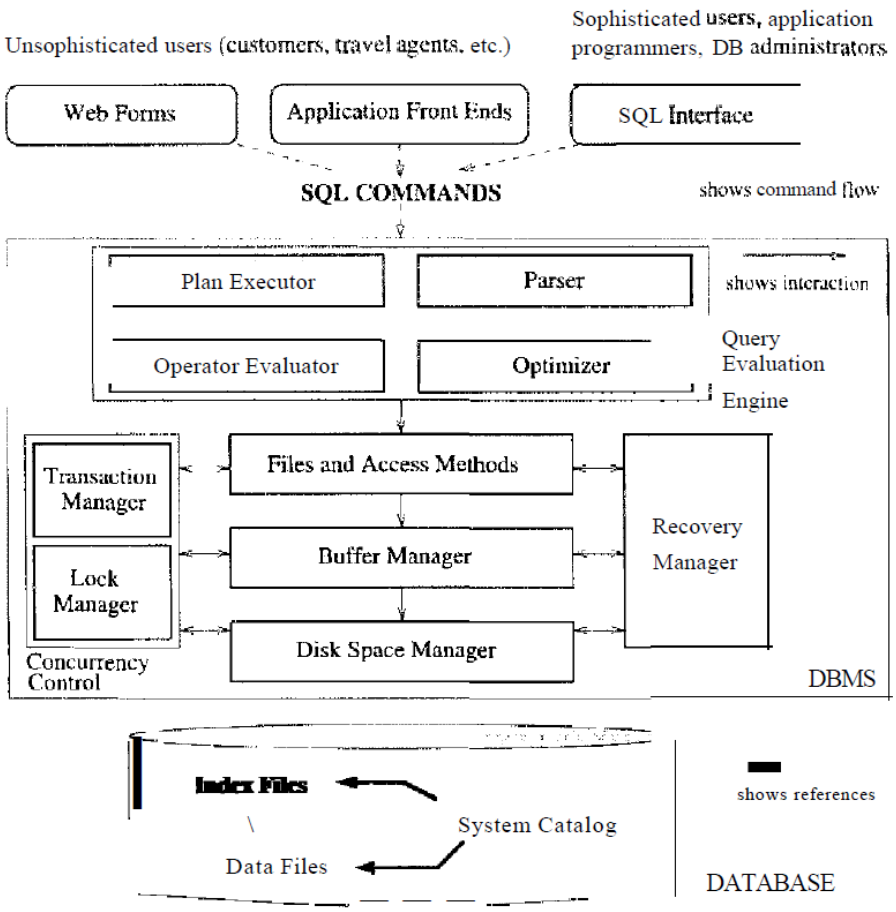
Second generation: Relational for Business App, 70s' from key and join to combine things

Third generation: Objects oriented, Object-relational increasing expressive power for Novel Applications

NoSQL for two reasons:

- change (increase or decrease) the expressive power of the data model, Increasing flexibility of the schema
- seek scalability.

it gives up some operations(like join) supported by the Relational DB. With certain app can be more optimal(because of specialization).The system may potentially be less optimized than a system with fixed schema. However, it will be able to support applications that couldn't have been supported otherwise (or would have been supported only with the great cost of having to recreate the database each time its schema is modified).



Storage and Indexing

Data in a DBMS is a collection of records, or a file, and each file consists of one or more pages. The files and access methods software layer organizes data carefully to support fast access.

A file organization is a method of arranging the records in a file. Each file organization makes certain operations efficient but other operations expensive.

Indexing can help when we have to access a collection of records in multiple ways, in addition to efficiently supporting various kinds of selection. Choosing a good collection of indexes is the most powerful tool for improving performance.

Disk

Data is stored on disks (sometimes on tapes).
Because: storing very lager data is cheap, addressing space is large, nonvolatile.

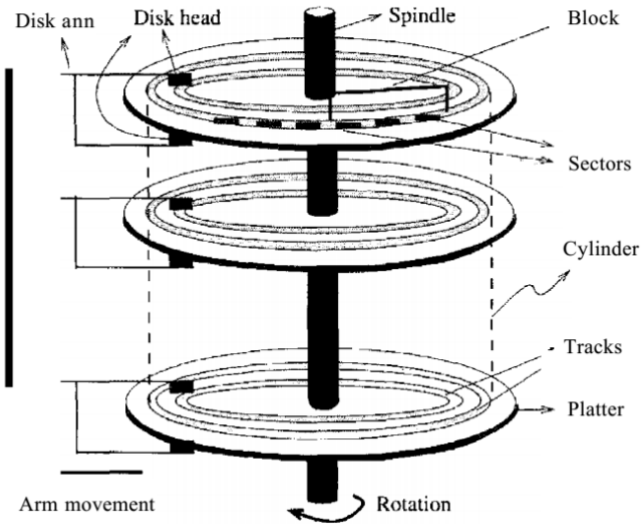


Figure 9.2 Structure of a Disk

- A disk has one or several platters.
- A platter is divided into several concentric rings called tracks. Tracks can be recorded on one(single-sided) or both surfaces(double-sided) of a platter.
- The set of all tracks with the Same diameter is called a cylinder.
- Each track is divided into arcs, called sectors, whose size is a characteristic of the disk and cannot be changed.
- Data is stored on disk in units called disk blocks. A disk block is a contiguous sequence of bytes and is the unit in which data is written to a disk and read from a disk. The size of a disk block can be set when the disk is initialized as a multiple of the sector size. Blocks are arranged in tracks.
- An array of disk heads, one per recorded surface, is moved as a unit. To read or

write a block, a disk head must be positioned on top of the block.

- A checksum is computed when data is written to a sector and computed again when the data on the sector is read back. If the sector is corrupted or the read is faulty, it is very unlikely that these checksums match. The controller computes checksums, and if it detects an error, it tries to read the sector again.

The time to access a disk block has several components:

- Seek time is the time taken to move the disk heads to the track on which a desired block is located.
- Rotational delay is the waiting time for the desired block to rotate under the disk head(half a rotation time).
- Transfer time is the time to actually read or write the data in the block(the time for the disk to rotate over the block).

Data must be in memory for the DBMS to operate on it. The unit for data transfer between disk and main memory is a block(in OS point of view, it is a page, whose size is 4 KB); if a single item on a block is needed, the entire block is transferred. Reading or writing a disk block is called an I/O operation. It dominates the cost of typical database.

The time to read or write a block varies, depending on the location of the data:

access time = seek time + rotational delay + transfer time

transfer time is small compared to S and R. Reduce access time = Reduce S and R.

We can do it

- By hardware. Increasing rotation speed and arm assembly moving speed, which is hard. RAID can also help.
- By software. Using sequential access as many as possible, which is faster than random access.

Because, we only pay S and R once.

We have `Next' block concept: blocks on same track, followed by blocks on same cylinder, followed by blocks on adjacent cylinder.

We can achieve that by:

- Blocks in a file should be arranged sequentially on disk
- For a sequential scan, pre-fetching several pages at a time
- Clustering things that are co-related in one page, according to row-base storage or column-base storage.

Files in DBMS

We have three kinds of files in a DBMS: catalog, data, index.

Catalog

- Catalog is itself a collection of tables.
- Catalog describes all the tables in the database, including the catalog tables themselves.
- The code that retrieves the schema of a catalog table must be handled specially.

- Catalog tables can be queried just like any other table.
- All the techniques available for implementing and managing tables apply directly to catalog tables.

Data

A data file is a collection of pages, each containing a collection of records. Each record in a file has a unique identifier(rid). An rid has the property that we can identify the disk address of the page containing the record by using the rid.

For a data file we can store it by a unorder file(heap file) or by a sorted file.

Sorted File

In a sorted file. every record and every page are consecutively stored in some order. It provides two advantages:

1. order-> search is cheap(for binary is $\log_2 N$, If want a higher # than 2, need indexing).
2. sequential(can pre-fetching) -> efficient scanning and range search.

Used for static data. Because update the file is very expensive(need shifting). We can use several techniques to mitigate it:

- deletion mark for deletion, use a mark to indicate this record is invalid.
- overflow page for insertion and lazily update, allocate a temporary page to hold new data and shift later on. overflow page becomes random access.

Heap File

Simplest file structure contains records in no particular order. As file grows and shrinks, disk pages are allocated and de-allocated. Thus, they are not necessarily consecutive.

To support record level operations, we must:

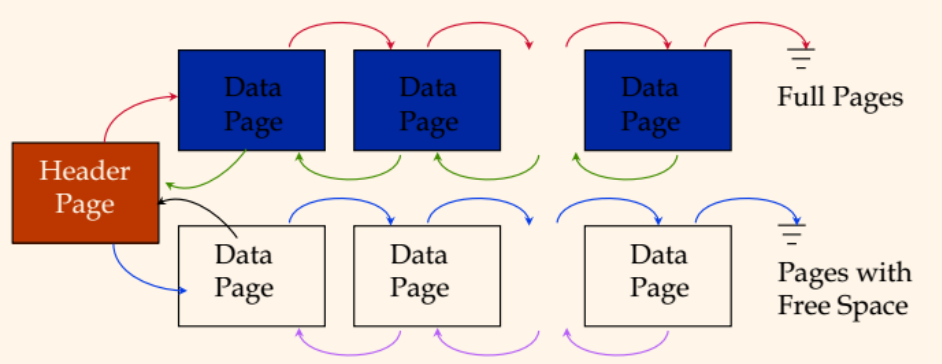
- keep track of the pages in a file(for scan)
- keep track of pages having free space (for efficient insertion)
- keep track of the records on a page(search, insert or delete)

We can use either a list or a directory to address the first two concerns. In each of these alternatives, pages must hold two pointers (which are page ids) for file-level bookkeeping in addition to the data.

List

One possibility is to maintain a heap file as a doubly linked list of pages. The DBMS can remember where the first page is located by maintaining a table containing pairs of (heap_file_name, page_addr) in a known location on disk.

We call the first page of the file the header page.

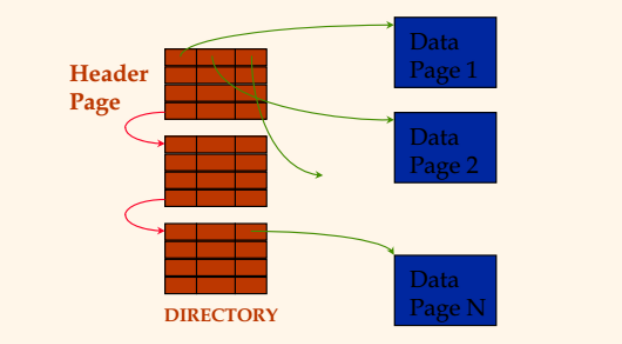


One disadvantage is that virtually all pages in a file will be on the free list if records are of variable length. To insert a typical record, we must retrieve and examine several pages on the free list before we find one with enough free space.

The directory-based heap file organization that we discuss next addresses this problem.

Directory

An alternative to a linked list of pages is to maintain a directory of pages. The DBMS must remember where the first directory page of each heap file is located. The directory is itself a collection of pages and is shown as a linked list.

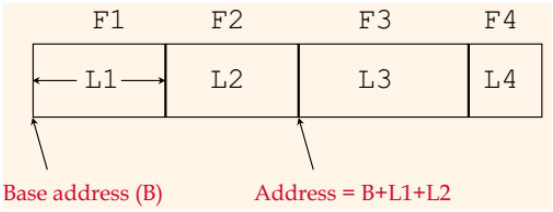


Free space can be managed by maintaining a bit per entry, or a count per entry. Since several entries fit on a directory page, we can efficiently search for a data page with enough space to hold a record to be inserted(Directory is relatively small, access less pages).

How to address the third concern depends on two cases. rid here is a pair <page id, slot number>.

Fixed-Length Records

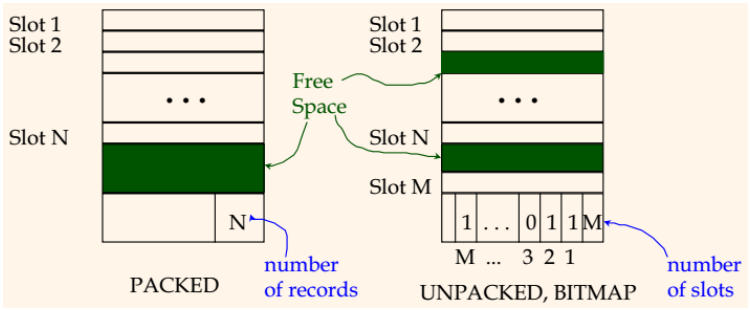
In a fixed-length record, each field has a fixed length, and the number of fields is also fixed. Given the address of the record, the address of a particular field can be calculated using information about the lengths of preceding fields, which is available in the system catalog.



If all records on the page are guaranteed to be of the same length, record slots are uniform and can be arranged consecutively within a page. There are two alternatives to keep track of empty slots and locate all records on a page.

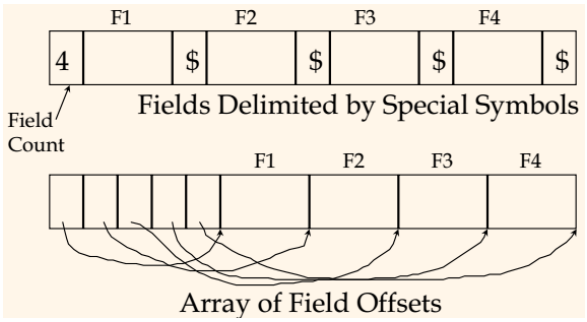
The first alternative is to store records in the first N slots (where N is the number of records on the page); whenever a record is deleted, we move the last record on the page into the vacated slot. This format allows us to locate the ith record on a page by a simple offset calculation, and all empty slots appear together at the end of the page. However, this approach does not work if there are external references(index, log file or other utilities) to the record that is moved (because the rid is changed). It is still useful if deletion is rare or do reformat in a batch way.

The second alternative is to handle deletions by using an array of bits, one per slot, to keep track of free slot information. Locating records on the page requires scanning the bit array to find slots whose bit is on; when a record is deleted, its bit is turned off.



Variable-Length Records

If the number of fields is fixed, a record is of variable length only because some of its fields are of variable length. How to distinguish each field.



One possible organization is to store fields consecutively, separated by delimiters (which are special characters that do not appear in the data itself). This organization requires a scan of the record to locate a desired field.

An alternative is to reserve some space at the beginning of a record for use as an array

of integer offsets-the ith integer in this array is the starting address of the ith field value relative to the start of the record. Note that we also store an offset to the end of the record; this offset is needed to recognize where the last field ends.

The second approach is typically superior. For the overhead of the offset array, we get direct access to any field. We also get a clean way to deal with null values. A null value is a special value used to denote that the value for a field is unavailable or inapplicable. If a field contains a null value, the pointer to the end of the field is set to be the same as the pointer to the beginning of the field.

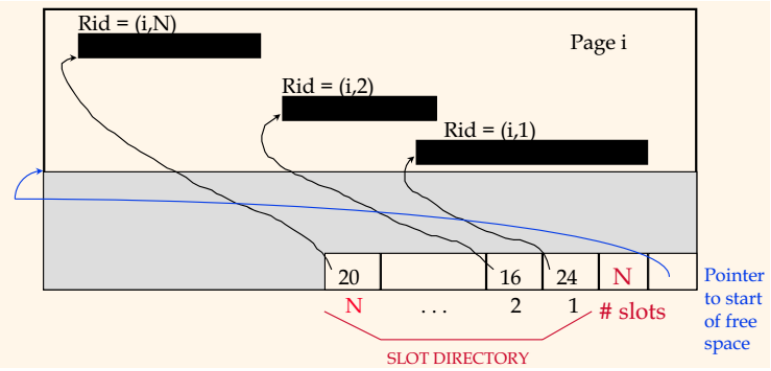
Variable-length record formats can be used to store fixed-length records for:

- **Supporting null values**
- **Adding extra fields**

Variable-length fields in a record can have issues:

- Modifying a field may cause it to grow, which requires shift.
- A modified record may no longer fit into the space remaining on its page. Moving to another page(rid may change) and its implications are hard to handle.
- A record may grow too large to fits on one page. breaking a record into smaller records chained together.

The most flexible organization for variable-length records is to maintain a directory of slots for each page, with a (offset, length) pair per slot. Deletion is readily accomplished by setting the record offset to -1. Records can be moved around on the page because the rid does not change.



One way to manage free space is to maintain a pointer that indicates the start of the free space area. When a new record is too large to fit into the remaining free space, we have to move records on the page to reclean the space freed by records deleted earlier. The idea is to ensure that, after reorganization, all records appear in contiguous order, followed by the available free space.

The slot for a deleted record cannot always be removed from the slot directory, because we may change (decrement) the slot number of subsequent slots in the slot directory. The only way to remove slots from the slot directory is to remove the last slot if the record that it points to is deleted. When a record is inserted, a new slot is added to the slot directory only if all existing slots point to records.

Some other salts

variable organization is also useful for fixed-length records if we need to move them around frequently(maintain sorted order). Also, we can store common length information once in the system catalog.

In some special situations, we do not care about changing the rid of a record. In this case, the slot directory can be compacted after every record deletion. We can also sort records on a page in an efficient manner by simply moving slot entries rather than actual records.

A simple variation on the slotted organization is to maintain only record offsets in the slots. for variable-length records, the length is then stored with the record. This variation makes the slot directory structure for pages with fixed-length records the same as for pages with variable-length records.

Indexing

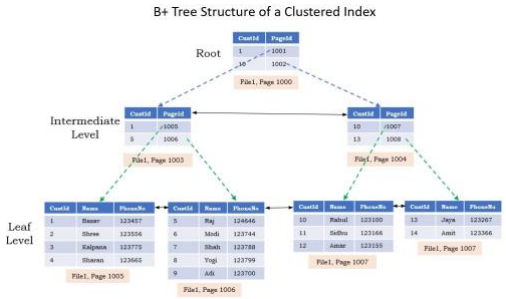
We use the term data entry to refer to the records stored in an index file. A data entry with search key value k, denoted as k*, contains enough information to locate data records with search key value k. We can efficiently search an index to find the desired data entries.
There are three main alternatives for what to store as a data entry in an index:

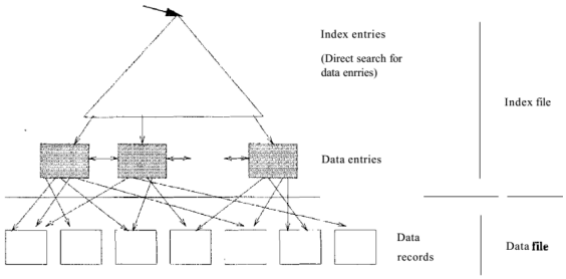
1) A data entry is an actual data record (with search key value k).
2) A data entry is a (k, rid) pair, where rid identifies a data record with search key value k.
3) A data entry is a (k. rid-list) pair, where rid-list is a list of record ids of data records with search key value k.

In case 1, We can think of such an index as a special file organization.
Case 2 and case 3, which contain data entries that point to data records, are independent of the file organization. 3) offers better space utilization than 2), but data entries are variable in length, depending on the number of data records with a given search key value.

Clustered Indexes

When a file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index, we say that the index is clustered. An index that uses 1) is clustered, by definition. 2) or 3) are usually unclustered, but can converted into a clustered one by sorting date file with some free space on each page for future inserts.





clustered	unclustered
1) means clustered	2) or 3) combined with heap file
At most one index(only one order)	Can have several
Data file changes with index file	Independent
Leaf nodes(data entries) are records	Data entries are pointers
one pointer per page(order within a page)	One pointer per record
Facilitate range query(traverse leaf nodes)	Facilitate index only query(traverse data entries)

The cost of using an index to answer a range search query can vary tremendously based on whether the index is clustered. If the index is clustered, the rids in qualifying data entries point to a contiguous collection of records, and we need to retrieve only a few data pages. If the index is unclustered, each qualifying data entry could contain a rid that points to a distinct data page, leading to as many data page I/Os.

Primary and Secondary Indexes

An index on a set of fields that includes the primary key is called a primary index; other indexes are called secondary indexes. Search key contains some candidate key, we call the index a unique index.

Composite Search Keys

The search key for an index can contain several fields; such keys are called composite search keys or concatenated keys.
To retrieve Emp records with age=30 AND sal=4000, an index on <age,sal> would be better than an index on age or an index on sal.
If condition is: 20<age<30 AND 3000<sal<5000:
Clustered tree index on <age,sal> or <sal,age> estimate unique values to choice the best one.
If condition is: age=30 AND 3000<sal<5000:
Clustered <age,sal> index will likely be better than <sal,age>(但如果整个数据库中都是年龄30 的人， 那么后者就会好)

Tree-structured

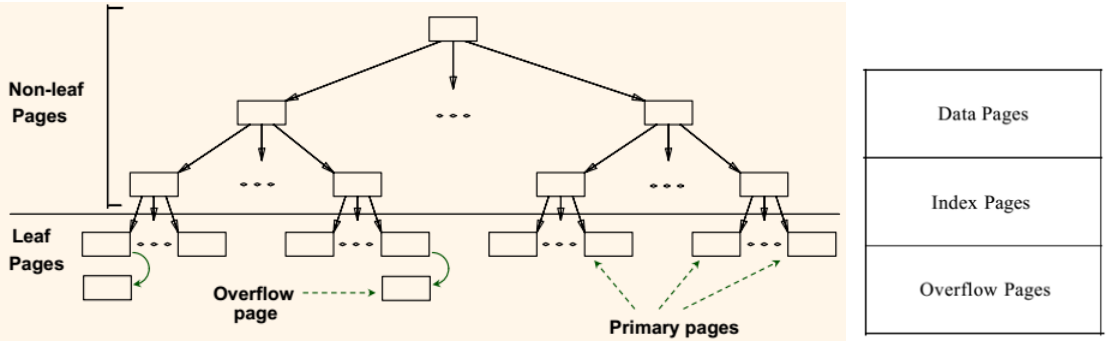
We now consider two index data structures, called ISAM and B+ trees, based on tree organizations. These structures provide efficient support for range searches, including sorted file scans as a special case. Unlike sorted files, these index structures support efficient insertion and deletion. They also provide support for equality selections, although they are not as efficient in this case as hash-based indexes.

An ISAM tree is a static index structure that is effective when the file is not frequently updated, but it is unsuitable for files that grow and shrink a lot. Whereas, a B+ tree is a dynamic structure that adjusts to changes in the file gracefully.

All non-leaf nodes are index entries, and contains m index entries and m + 1 pointers.

# of pages needed to be accessed are the major concern. We may need linear or binary search within a node during searching.

ISAM



Each tree node is a disk page, and all the data resides in the leaf pages. This corresponds to an index that uses Alternative (1) for data entries. We can create an index with Alternative (2) by storing the data records in a separate file and storing (key, rid) pairs in the leaf pages of the ISAM index.

When the file is created, all leaf pages are allocated sequentially and sorted on the search key value. The non-leaf level pages are then allocated. If there are several inserts to the file subsequently, additional pages are needed because the index structure is static. These additional pages are allocated from an overflow area.

- The basic operations of search is quite straightforward(like a search tree).
- For a range query, the starting point is determined by search, and data pages are then retrieved sequentially(may include overflow pages).
- For inserts and deletes, the appropriate page is determined by search, and the record is inserted or deleted with overflow pages added or deleted if necessary. If a primary page becomes empty, leave it empty to serve as a placeholder for future insertions.

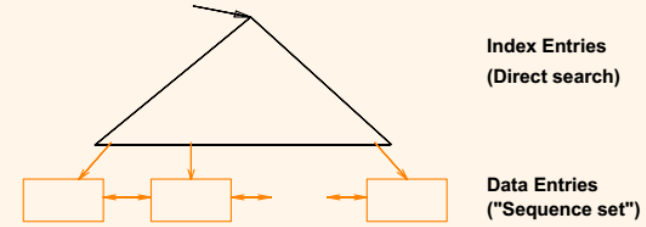
Note that, once the ISAM file is created, inserts and deletes affect only the contents of leaf pages. Thus, the number of primary leaf pages is fixed at file creation time. leads to:

- Long overflow chains could develop. These chains can significantly affect the time to retrieve a record. Overflow chains can be eliminated only by a complete reorganization

of the file. To alleviate this problem, the tree is initially created so that about 20 percent of each page is free.

- Not locking index-level pages. Facilitate concurrent accesses.（不会改变树的结构）
- leaf pages have no next page pointers, because they are always consecutive.(存储效率)
- index pages are fully utilized->max fan-out->max search speed.（存储效率->搜索效率）
- when data is static, range search is very efficient.

B+Tree



The B+ tree search structure is a balanced tree in which the leaf nodes contain the data entries. Since it is not feasible to allocate the leaf pages sequentially, we have to link them using page pointers.

dynamic data -> data entries not consecutive, no 100% page utilization-> more index file pages -> no max fan-out -> no mini height -> more page access when search

how to increase fan-out:

- compress Key values in index entries
- two physical consecutive pages mapped to one logic page
- eliminating duplicates and maintaining multiply pointers per key(need variable length records)

How to build b+tree:

- start with an empty tree and insert an entry one at a time. very expensive.
- bulk-loading, (1) creating the data entries to insert in the index, (2) sorting the data entries, and (3) building the index from the sorted entries.

bulk-loading has advantages:

- Has advantages for concurrency control.
- Fewer I/Os during build.
- Leaves will be stored sequentially (and linked, of course).
- Can control “fill factor” on pages.

Hash-based

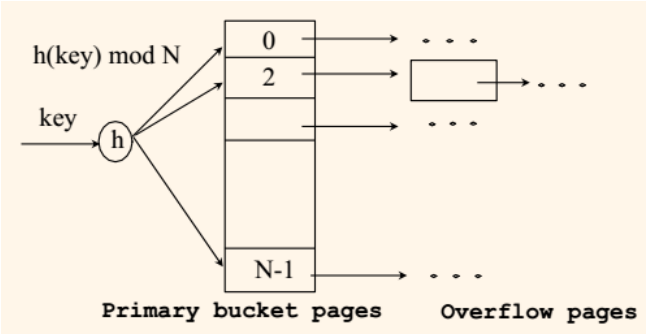
In this chapter we consider file organizations that are excellent for equality selections but not support range search. The basic idea is to use a hashing function, which maps values in a search field into a range of bucket numbers to find the page on which a desired data entry belongs. It must distribute values in the domain of the search field uniformly over the

collection of buckets to get a good performance.

static

The pages containing the data can be viewed as a collection of buckets, with one primary page and possibly additional overflow pages per bucket. A file consists of buckets 0 through N – 1(static means N is fixed), with one primary page per bucket initially.

- To search for a data entry, we apply a hash function h to identify the bucket.
- To insert a data entry, identify the correct bucket and then put the data entry there. If there is no space, allocate a new overflow page
- To delete a data entry, locate the data entry and then remove it. If this data entry is the last in an overflow page, the overflow page is removed.



The main problem with Static Hashing is that the number of buckets is fixed. If a file shrinks greatly, a lot of space is wasted; more important, if a file grows a lot, long overflow chains develop, resulting in poor performance.

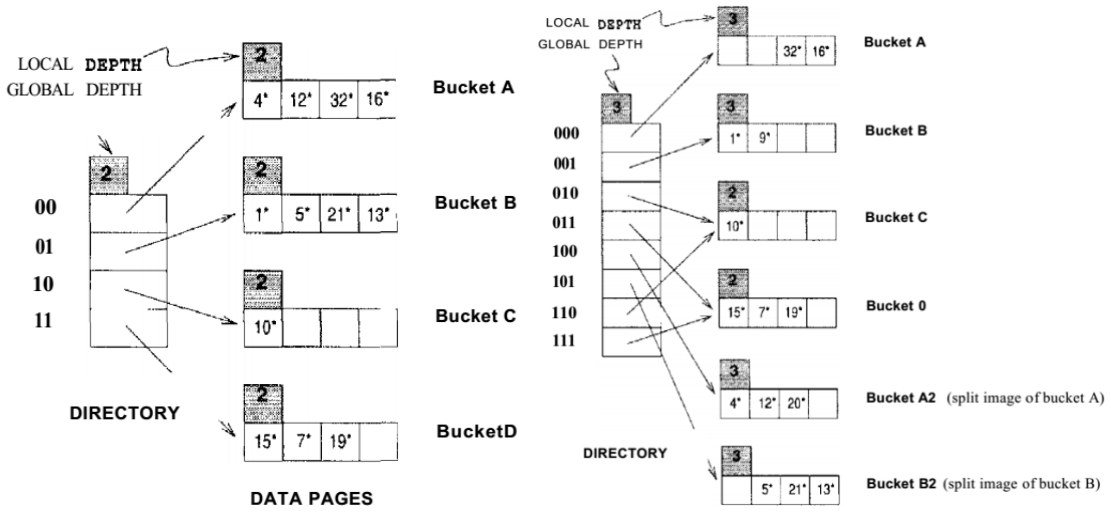
One simple alternative to Static Hashing is to periodically 'rehash' the file to restore the ideal situation (no overflow chains, about 80 percent occupancy). However, rehashing takes time and the index cannot be used while rehashing is in progress.

dynamic

dynamic hashing techniques can deal with inserts and deletes gracefully.

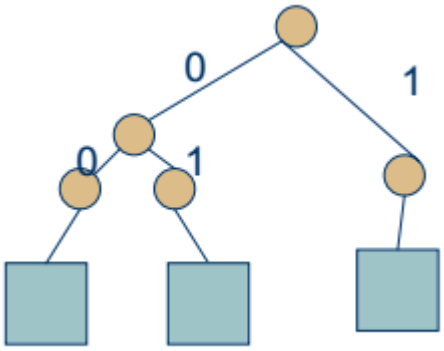
extensible

use the hash function's last d(global depth) digits to determine buckets.



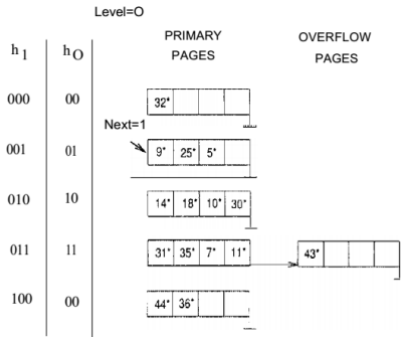
To determine whether a directory doubling is needed, we maintain a local depth for each bucket. If a bucket whose local depth is equal to the global depth is split, the directory must be doubled.

we can also use a prefix-based index structure such as tries



linear

Augment the hash table one slot at a time.
The scheme utilizes a family of hash functions h0, h1, h2, ... , with the property that each function's range is twice that of its predecessor.



level indicates which hash function we use first. If the number is above the pointer, we use

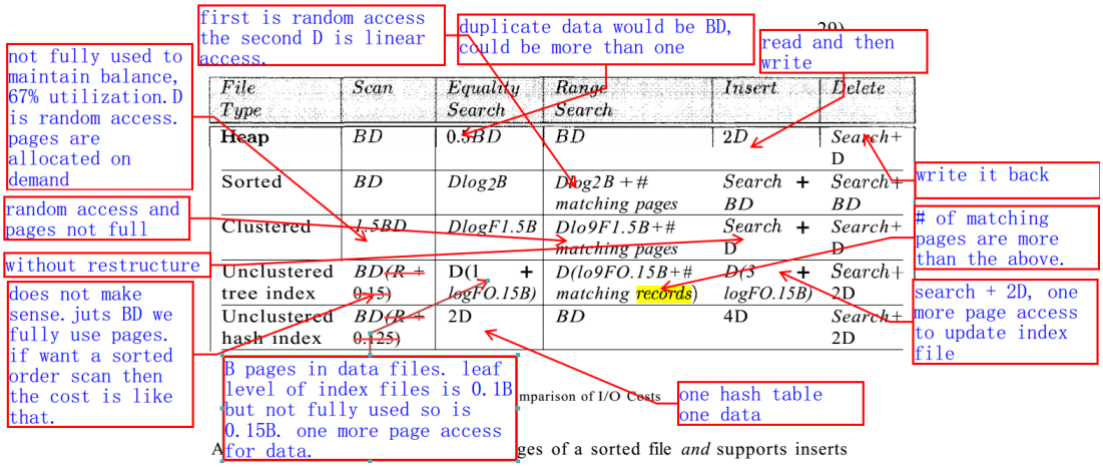
next level function. When to split the pages is arbitrary. Maybe when overall usage is 80%, or have some overflow pages. When the next is reach some point, we increase the level to use the next level function directly.
If the data distribution is very skewed, however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing.

Comparing File Organizations

We ignore CPU costs, ignores gains of pre-fetching for simplicity, Average-case analysis.
Measuring number of page I/O's a sequence of pages.

- B: The number of data pages
- R: Number of records per page
- D: (Average) time to read or write disk page

Heap Files: Equality selection on key; exactly one match.
Sorted Files: Files compacted after deletions.
data entry size = 10% size of record;
Indexes: Hash: No overflow buckets. 80% page occupancy => File size = 1.25 data size
Tree: 67% occupancy (this is typical), Implies file size = 1.5 data size



File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5BD	BD	2D	Search+ D
Sorted	BD	Dlog2B	Dlog2B +# matching pages	Search + BD	Search+ BD
Clustered	1.5BD	DlogF1.5B	Dlo9F1.5B+# matching pages	Search + D	Search+ D
Unclustered tree index	BD(R + 0.15)	D(1 + logFO.15B)	D(lo9FO.15B+# matching records)	D(3 + logFO.15B)	Search+ 2D
Unclustered hash index	BD(R + 0.125)	2D	BD	4D	Search+ 2D

mparison of I/O Costs
ges of a sorted file *and* supports inserts

Utilization of pages affect overhead.
In tree-structure clustered index: index file changes in step with data file.
In tree-structure unclustered index: index file is independent of data file.
index page utilization -> fanout -> search -> many related operations.
data page utilization -> file size -> scan and range search

Disk Space Management

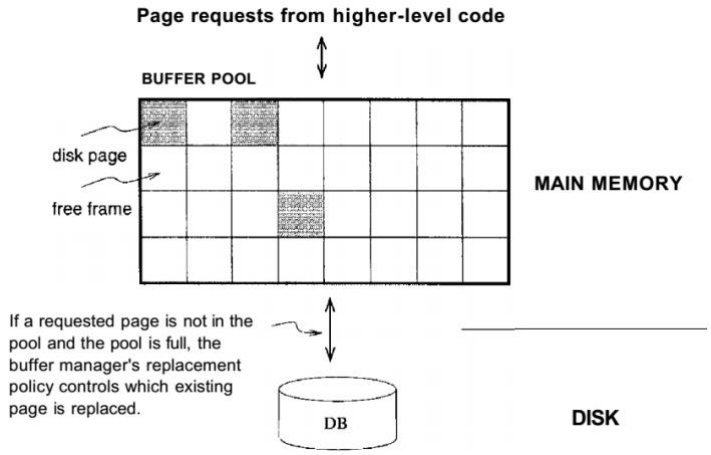
Space on disk is managed by the disk space manager. Lowest layer of DBMS software manages space on disk. Higher levels call upon this layer to: allocate/de-allocate a page; read/write a page.
The disk space manager keeps track of the pages in use by the file layer; if a page is freed by

the file layer, the space manager tracks this, and reuses the space if the file layer requests a new page later on.
Request for a sequence of pages must be satisfied by allocating the pages sequentially on disk!

Buffer Manager

buffer manager is the software layer responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the buffer pool. The main memory pages in the buffer pool are called frames.

In addition to the buffer pool itself, the buffer manager maintains some bookkeeping information and two variables for each frame in the pool: pin_count and dirty. The number of times that the page currently in a given frame has been requested but not released-the number of current users of the page—is recorded in the pin_count variable for that frame. The Boolean variable dirty indicates whether the page have been modified since it was brought into the buffer pool from disk.



Initially, the pin_count for every frame is set to 0, and the dirty bits are turned off. When a page is requested the buffer manager does the following:
1. Checks the buffer pool, if have, increments the pin_count. If not, the buffer manager do as follows:
(a) Chooses a frame for replacement and increments its pin_count.
(b) If the dirty bit is on, writes the page to the disk.
(c) Reads the page into the frame.
2. Returns the address of the frame.

Incrementing pin_count is called pinning the requested page in its frame. (a pages is actively used by some queries or transactions, until commit) Decrementing the pin_count of the frame is called unpinning the page. If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the dirty bit for the frame is set.

If a requested page is not in the buffer pool and a free frame is not available in the buffer pool, a frame with pin-count 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager's replacement policy.

When a page is eventually chosen for replacement, if the dirty bit is not set, there is no need to write the page back to disk. Otherwise, the modifications to the page must be propagated to the copy on disk.

If no page in the buffer pool has pin_count 0 and a page that is not in the pool is requested, the buffer manager must wait until some page is released before responding to the page request.

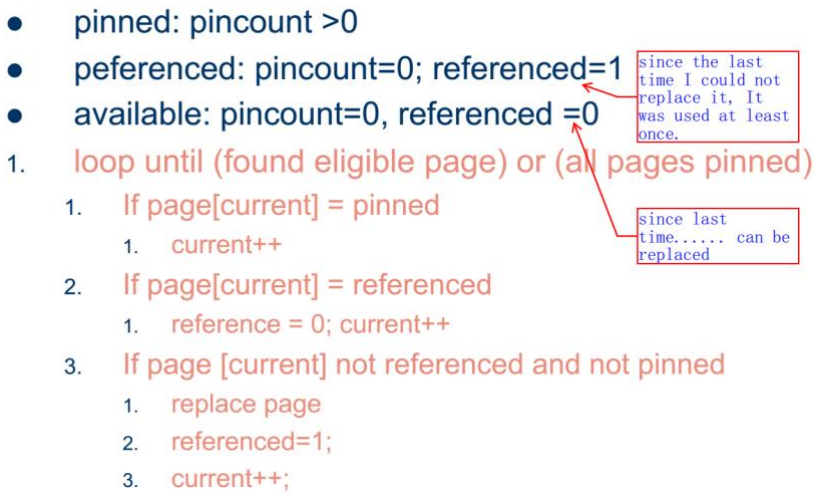
If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time

Replacement Policy:

The policy used to choose an unpinned page for replacement can affect the time taken for database operations considerably. Of the many alternative policies, each is suitable in different situations.

The best-known replacement policy is least recently used (LRU).

A variant of LRU, called **clock** replacement, has similar behavior but less overhead.



The LRU and clock policies(with assumption temporary locality) are not always the best replacement strategies for a database system, particularly if many requests require sequential scans of the data. Such situation is called sequential flooding: # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

Other replacement policies include first in first out (FIFO) and most recently used (MRU), which also entail overhead similar to LRU, and random. LFU, frequency may not be useful, and we add more overheads.

Designing BMSs that understand DBMSs

DBMSs [sometimes] know access pattern (regular and predictable) due to a limited set of operations. It can sometimes figure out how much memory it needs for an operation as a reference pattern of a database operation can be decomposed into a number of simple reference patterns.

Domain Separation

Domain Separation (DS) Buffer Replacement Algorithm with a B-tree index (aka page-type-oriented buffer allocation scheme)
Observation: root pages of indexes are accessed more often.
Pages classified into types with an associated domain of buffers. In case no buffer left, borrow from another task. Buffers inside each domain managed by LRU.
A possible scenario: One domain to each non-leaf level of the B-tree structure, and one to the leaf level together with the data. provided 8-10% improvement in throughput comparing with an LRU.
Problems:

- Domain is static; Fails to capture query dynamicity. Some queries are more frequent than others thus, some pages are more popular.
- Does not differentiate between different pages. Index pages could be more important that data pages.
- Partitioning buffers according to domains, rather than queries, does not prevent interference among competing users.
- a separate mechanism needs to be incorporated to prevent thrashing.

Extension:

- GLRU: there exists a fixed priority ranking among different groups (domains). A search for a free buffer always starts from the group with the lowest priority.
- dynamically vary the size of each domain using a working-set-like partitioning scheme. The “working set” of each domain may grow or shrink depending on the reference behavior of the user queries.

NEW

The buffer pool is subdivided and allocated on a per-relation basis.
Observations:

- The priority for a page is a property of the relation to which it belongs rather than the page itself.
- Each relation needs a “ working set ”.

Each resident set associated with an active relation:

- initially empty
- linked in a priority list with a global free list on the top and the resident set whose pages are unlikely to be reused placed near the top.
- replace with MRU
- Each relation is entitled to one active buffer which is exempt from replacement consideration.

Pros

- Track the locality of a query through relations.

Cons

- MRU is justifiable in limited cases.

- The rules for arranging the order of relations were based solely on intuition.
- Finding a free buffer in the priority list can be expensive under high memory contention.
- It is hard to extend the algo for multi-users with respect to priority assignment for relations from different concurrent queries.

Hot set

A query behavior model for relational database systems that integrates advance knowledge on reference patterns into the model.
A hot set: is a set of pages over which there is a looping behavior.
A hot point: is a discontinuity around the buffer size in the curve where the number of page faults is a function of buffer size.
Ex.: For a nested loops join, the number of pages in the inner relation plus one for the outer relation. If the scan on the outer relation is an index scan, an additional buffer is required for the leaf pages of the index.(Or there will be a lot of swaps)
Each query is provided a separate list of buffers managed by an LRU discipline. The number of buffers equals to the query's hot set size. A new query whose hot set size exceed the available buffer space will not be allowed to enter the system.
Cons:

- The derivation of the hot set model is based partially on LRU, which is inappropriate for certain looping behavior. In fact, MRU, which retains older data, is more suitable for cyclic access patterns. Thus, the hot set model does not truly reflect the inherent behavior of some reference patterns, but rather the behavior under an LRU algo.
- Low memory utilization due to over-allocation for that a query referencing to different data structures will not interfere with another.

DBMIN

Query Locality Set Model (QLSM)

Predict future reference behavior so that has better performance than the stochastic models. Separate the modeling of reference behavior from any particular buffer management algo, which is the potential problems of the hot set model.

Sequential patterns

Straight Sequential (SS)
ex: file scan for select operations
one page is enough; do not have to kill others
replace with next
Prefetching: cache in disk controller, good for SS
Clustered Sequential (CS)

ex: inner relation for sort-merge join
pages in the largest cluster (statistical result).

Looping Sequential (LS)

ex: inner relation in Nested Loops Join (not required for sort)
use MRU & allocate as many pages as possible (up to the inner file size)

Random References

Independent Random (IR)

a series of independent random accesses
ex: an index scan through a non-clustered index
ex: scan through a hash table
one page if page references are sparse, more that one is not guaranteed to be useful.
the choice of a replacement algo is immaterial since all perform equally well [King71]

Clustered Random (CR)

a series of " random " accesses in a " cluster " that are not physically adjacent but randomly distributed over the file.
ex: a join where the inner relation uses a non-clustered and non-unique index while the outer relation is a clustered file with non-unique keys.

Hierarchical References

a sequence of page accesses that form a traversal path from the root down to the leaves of an index

Straight hierarchical (SH)

traverse the index only once
ex: retrieving a single tuple
one page is enough for buffering all the index pages

Hierarchical with straight sequential (H/SS)(using SS strategy)

a tree traversal followed by a Straight Sequential scan through the leaves one page.

Hierarchical with clustered sequential (H/CS)(using CS strategy)

a tree traversal followed by a Clustered Sequential scan through the leaves

Looping Hierarchical (LH)

ex: repeated accesses to the index pages in a join where the inner relation is indexed on the join field
Pages closer to the root are more likely to be accessed than those closer to the leaves.
A priority list where pages closer to the root have higher priority than those closer to the leaves. In many cases, the root is perhaps the only page worth keeping in memory due to high fan-out of an index page.

DBMIN

- buffers allocated on a per file instance basis. Active instances of the same file are given different buffer pools with independent replacement policy, but all the file instances

share the same copy of a buffered page whenever possible through a global table mechanism.

- each file instance has a local, separately managed buffer pool to hold its locality set, the size of which is determined in advance and needs not be re-calculated during the query. And the file instance is considered the owner of all the pages in its locality set.
- a global, shared table of buffers, with LRU policy, for allowing data sharing among concurrent queries.

Algorithm:

1. Page found in global table and in the requested locality set, I
   - Use local replacement policy of I
2. Page found in the global table, but not in the locality set
   - Page has an owner: give the page to the new request
   - Page has no owner: put the page in locality set, I (may need replacement)
3. Page not found
   - Bring it to the main memory and proceed as 2.

Advantage:

- When replaced, pages don't necessarily swap to disk. Mark it as free so it's still in memory until kicked out by someone else. So can get it back without having to read the disk. High chance will find it in main memory.
- Can leverage diverse knowledge about tables, access patterns, and queries. Only a hand full of access patterns are enough

Query Processing

Relational operators and their evaluations

Relational operators serve as building blocks for evaluating queries, and the implementation of these operators is carefully optimized for good performance.

The algorithms for various relational operators actually have a lot in common. A few simple techniques are used to develop algorithms for each operator:

- Indexing: If a selection or join condition is specified, use an index to examine just the tuples that satisfy the condition.
- Iteration: Examine all tuples in an input table, one after the other. If we need only a few fields from each tuple and there is an index whose key contains all these fields, instead of examining data tuples, we can scan all index data entries.
- Partitioning: By partitioning tuples on a sort key, we can often decompose an operation into a less expensive collection of operations on partitions. Sorting and hashing are two commonly used partitioning techniques.

Some common relational algebra operators are : Selection, Projection, Join, Set Operations and so on. The standard relational operations stand for **logical operators**. There are several alternative algorithms are available for implementing each logical relational operator. The actually implementation is **physical operators**.

Operations can be classified into pipelined, and blocking, non-blocking.

pipelined access: access data with getnext(). Before the operation is finished for the whole data, we can get the first data as soon as possible, like lazily computing. It has the great virtue of not writing the result of intermediate joins to a temporary file because the results are produced, consumed, and discarded one page at a time. If the algorithm implemented for the operator allows input tuples to be processed completely when they are received, input tuples are not materialized and the evaluation is pipelined.

Getting the first result fast. crash recovery is hard.

non-pipelined is also called batch access(like Hadoop). We cannot get the results until operations is done. But recovery is easy.

blocking access: It can block the pipeline. doing operations consuming all of the data from the input side. after finishing all the computation, can product the results. blocking is not desired. as have to wait, or only need a few results.

Selection, Projection versus Scan:

Both selection and projection can be implemented using a scan operator.

Scan is a physical operator, there are two kind of scan:

File Scan: Going through every tuples in a data file, then output the tuples that satisfy the specified conditions and/ or the attributes list.

Index file Scan: Using an appropriate index to find the tuples that satisfy the conditions. could do index only scan if we only project the index part.

For projection, we sometimes need distinct results. Then duplicate elimination comes handy.

Duplicate Elimination: eliminating duplicate tuples which is an expensive operation. There are three ways to do that:

- Sorting and then removing duplicates. (Can drop unwanted information while sorting.)
- Hashing to create partitions. Load partitions into memory one at a time, and eliminate duplicates.
- Using index, If an index contains the desired fields, we could use index-only scan to get these data.

Sort

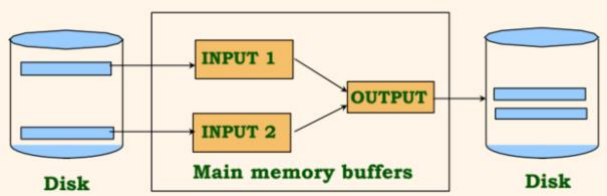
Sorting is necessary for some queries and make some operations cheaper, like duplicate elimination and group by, sort-merge, bulk loading in B+-tree. Even if a query do not need it, we sometimes still do it to reduce cost. Sorting is a blocking operation.

Merge sort in DBMS

Quick sort does not used in DBMS, because it moves data around, which involves a lot of random accesses.

2-Way Merge Sort

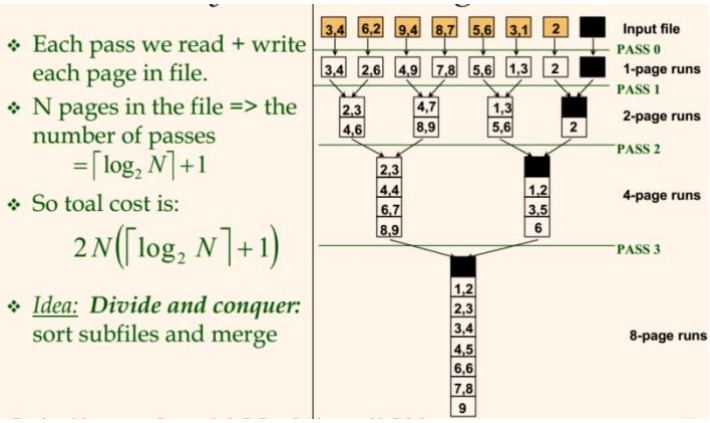
It use only three buffer pages.



Pass 0: split the data into pages sort each and store. each is a sort run. The whole operation is a scan.

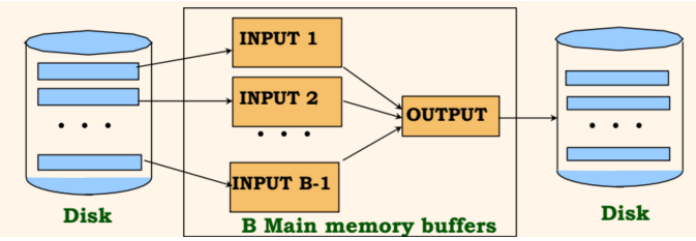
Pass 1: two buffer pages to read and another one to write. The operation is basically scan the two input buffers and output the results in order.

Pass 2: just like a normal merge sort



General External Merge Sort

If we have more than three buffer pages, then



In this case,

- Number of passes: $1+\lceil \log_{B-1}\lceil N/B \rceil \rceil$
- Cost = 2N * (# of passes)

The pass 0, use B pages of buffer to generate Ceiling(N/B) groups of data.

Using heapsort as the internal sort algorithm

Using heapsort can facilitate the initial run to generate Ceiling(N/(2 * B)) groups of data.

In fact, we need B + 2 pages, as one is for input buffer and the other is for output buffer. For

simplicity, we omit that. Suppose we have a four-page buffer.

- Read four pages into buffer, cannot read more ,then output data.
- output one, take another one in. repeat until encounter a counter that cannot make the output run sorted anymore (which means the heap output a tuples that is larger than the front tuple of the output), then output everything in the heap.
- Repeating the whole process.

Pros: Usually B buffer frames will have 2B length run.
Cons: length of each run will vary. hard to implement.

Join

Join is common in DBMS, if we implement it by a Cartesian product followed by a selection, it will be very inefficient.
Assume: M pages in R, Pr tuples per page, N pages in S, Ps tuples per page.
Only equality join.
Cost metric: # of I/Os. We will ignore output costs.

Nested loops join(Non-blocking)

Simple one

It only needs two buffer pages.
Tuple-orientated:
For each tuple in the outer relation R, we scan the entire inner relation S.
Cost: M + Pr * M * N
If we change order, outer is S and inner is R, then
Cost: N + Ps * M * N
So, we prefer to put the relation having large records (P is small) in the outer.
Page-orientated:
For all tuples in each page of the outer relation R, we scan the entire inner relation S.
Cost: M + M * N
After change order,
Cost: N + M * N
Then, put the relation having less amount of pages in outer.

Use index

If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
Cost: M + M * Pr * cost of finding matching S tuples
For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of

then finding S tuples depends on clustering. Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Block one

Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
Cost: M + ceiling (M / # of blocks) * N
With sequential reads considered, analysis changes: may be best to divide buffers evenly between R and S. It also could be that give one for the outer and all the other to the inner.
But if the buffer is large enough(MIN(M + 1, N + 1)) to hold the entire inner or outer, the cost is M + N.

Sort-Merge Join(If need sort, blocking)

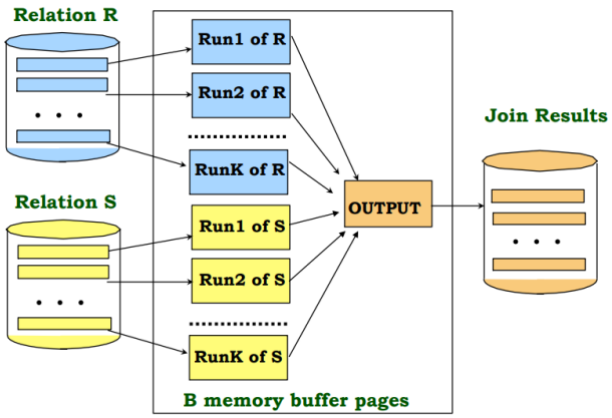
Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples. This is symmetrical for outer and inner.

- Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
- At this point, all R tuples with same value in Ri (current R group) and all S tuples with same value in Sj (current S group) match; output <r, s> for all pairs of such tuples(Cartesian Product).
- Then resume scanning R and S. (We may want to move the pointer which points to the next one that is the smallest among the two sides.)

Cost: M log M + N log N + (M+N), The cost of scanning, M+N, could be M*N (all are the same is very unlikely!)

Refinement(Two-pass merge sort)

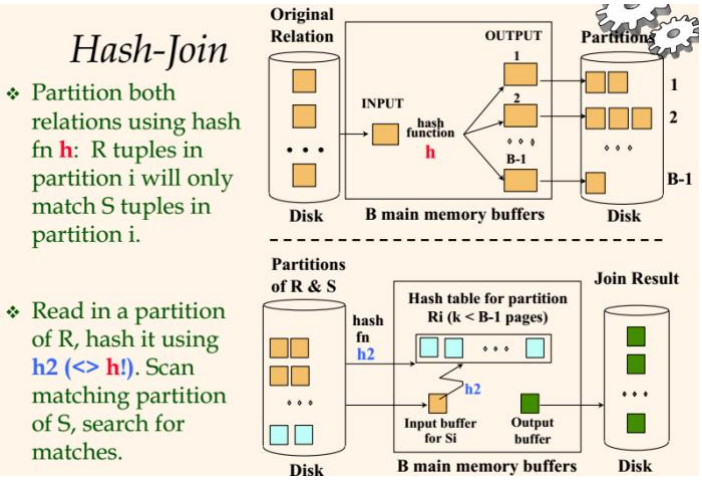
We can combine the merging phases in the sorting of R and S with the merging required for the join.



- With B > sqrt(L) (that is # of runs * B > L), where L is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, # of runs of each relation is < B/2.
- Allocate 1 page per run of each relation, and `merge' while checking the join condition.

Cost: read and write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples, ignore here) = 3 (M + N), is linear.

Hash-Join(partitioning cause blocking)



two relations are hash partitioned with B buffer pages.

one page used as input, then B-1 as output buffer. if one buffer is full, write it to the disk.

use the same hash function h(). The tuples that are similar to each other based on the condition could be in a certain partition.

- #partitions k < B-1 (why?), and B-2 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - k= B-1, and M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

pay the price to create the partitions. cost depends on B.

if the entire relation has the same tuples, no benefit from hash, degrade to Cartesian product.

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Query plans and optimizations

Query plan

A plan to execute a query. We usually convert it into a tree called query tree.
A logical query plan involves standard relational operators, just shows what to do, does not specify how to do a join for example. need to converted into a physical query plan.
A physical plan shows how to do, every node is a physical operator. Some logical operator may disappear and some new physical operators may be introduced. It specifies not only which algorithm to use but also how many buffers used for each individual task.
For a join operation, the order does matter in a physical join. but not in a logical join as the results are the same.
Three distinct query plans(trees):

- left deep(not only pipelined but also can leverage all indexes)
- right deep (most RDB avoid this one. 1. index usage, can only use index on the inner side, so index is useless here 2. need materialize the inner side then can proceed, (blocking) the left deep is not blocking, as the inner is ready to be access)
- bushy (chance to be parallel, but also partial blocking and leverage partial index, single machine will not consider this)

An access path is a way of retrieving tuples from a table and consists of either (1) a file scan or (2) an index plus a matching selection condition. Every relational operator accepts one or more tables as input, and the access methods used to retrieve tuples contribute significantly to the cost of the operator.
The **selectivity** of an access path is the number of pages retrieved (index pages plus data pages) if we use this access path to retrieve all desired tuples.
Choice the most selective path, require less amount of page accesses.
The fraction of tuples in the table that satisfy a given conjunct is called the **reduction factor**.
For multiples conditions, a commons technique is used to convert them into conjunctive normal form, like (| | |)&(| | |) simplify a query planning. easy to estimate cost. tasks become standard. can always do that.

Query optimization

In general, queries are composed of several operators, and the algorithms for individual operators can be combined in many ways to evaluate a query. The process of finding a good evaluation plan is called query optimization.

The optimization can be viewed as a search problem. Given a SQL query, get relational calculus is trivial, finding the correspond Relational operations is also trivial, finding some other logical equivalencies is also simple. The problem is every plan has a cost. how to find the optimal or sub-optimal plan. A search problem.
Then we need:

- A query plan enumerator.

- Statistics about data and index. It is the input to cost model. We can only maintain some statistics, because it is expensive to maintain all kinds of information(exponential complexity). Then hard to estimate correctly, when conditions are very complex. like a selection operator can have many kinds of conditions or their combinations.
- A cost model. the parameters could be # of tuples, # of pages, unique keys or values, property of index such as height. (take a one query plan and output the page access) how precisely a cost model is? it depends. Do not know the buffer could be a problem.
- A search algorithm to approach the goal. Given both model and statistics are not precise, then the so called optimal is not optimal. Then we avoid bad query plans. Assuming the worse cases(like no buffer) happen then can avoid.

Two method: heuristic approach and System R style(cost based). usually used together.

heuristic-based query optimization:

perform cheaper operations first. cheaper operations eliminate some of the inputs, hence more expensive operations need to deal with a smaller number of inputs.
both selection and projection reduce table(improve buffer efficiency). one in col the other in row.
like selection first then join, not join first, then selection.(push the selection down)
so can projection first then join. (push the projection down)
Challenge: This assumes that we can freely change the order of operations (associatively, commutativity)

System R

It is the first cost-based, also almost in all modern query optimizers.
Construct a query instead of search. it is optimal under certain assumptions.
assumption: we count I/O total cost = sum(all cost for each operation)
For a query tree, we not only need to know the cost model of each operation, but also need a model to estimate the size of intermediate relations.
B(R) <- number of blocks
T(R) <- number of tuples
V(R.a) <- number of distinct values for an attribute “a”
V(R.[a1, ..., an] <- number of distinct values when a1-an are considered together.

Scan

A scan operator. input could be conditions and attributes list or no constrain.
Just take look at the size of the table, NULL condition and projection will be the same #.
For selection, with one condition A. the # of tuples will be based on the selectivity of the condition.
max: N-V(A) + 1 , min: 0, average: N/V(A)(we may maintain a histogram to get a more

accurate estimation but that is expensive)
If we have two conditions. A and B, we often assume A and B are independent(or need to use histograms). we can treat their selectivity as probabilities. then A ^ B = 1/V(A) * 1/V(B).
If it is OR between the two conditions, treat them as probability again, also assume independent. If they are dependent, use histogram.
If the condition is like < or >, could use histograms, or according to the distinct values distribution.

Join

R(x,y) S(y,z)

- T(R |><| S) = 0, no matching condition
- T(R |><| S) = T(R) foreign key
- T(R |><| S) = T(R)T(S) Cartesian Product
- T(R |><| S) = $\frac{T(R)T(S)}{\max(V(R,y),V(S,y))} = \frac{MIN(V(R,y),V(S,y))T(R)T(S)}{V(R,y),V(S,y)}$

In the first equation, we assume all the unique values in one side are subsumed in the other side, values are uniform distributed. Like:
1 2 3 4 5
1 2 3
Then all tuples in the side which has less # of distinct tuples will join with the group in the other side. Like 1 will join with all the tuples have 1 on the other side. The selectivity is $1/\max(V(R,y),V(S,y))$. One fifth in that case.
In the second equation, we start from Cartesian product T(R)T(S). Then eliminate tuples that are not equal to a certain value y, $\frac{T(R)T(S)}{V(R,y),V(S,y)}$, Then multiply $MIN(V(R,y),V(S,y))$ the # of distinct values in the results. This kind of question may appear in final exam!
R(x,y1, y2) S(y1, y2,z)

- T(R |><| S) = $\frac{T(R)T(S)}{\max(V(R,y1)V(S,y1))\max(V(R,y2),V(S,y2))}$ two columns join

Before we construct our query plan we have two assumptions:
Projections and Selections are already pushed down. Then join order matters.
Only consider left-deep. because use index and non-blocking.
Suppose have 4 relations then 3 joins, have 4! ways to join.
suppose we fix R4 to the last one to join, then we can have 3! query plans.
Each one has a cost and # of results. costs could be different, but results are the same(order maybe different).
Then the last step's cost is fixed. select the cheapest one.
That is Principle of sub-query optimality.
However, that maybe is not a good idea. Because if someone gives an interesting order one.
Final join is on A, then we can use sort-merge join.
Suppose we choose the cheapest one, use recursion.
Problem: NP-hard problem. Can eliminate redundant works. Using and top-down approach,

memorization(overhead: look ups and maintain) or an bottom-up approach, DP, solve small problems first, then construct.
This DP algorithm acts as the starting point for other algorithms.
In practical, we also maintain multiple interesting-order plans.
If we change the order, some joins will become Cartesian product, we may omit them directly.

Transaction Management

Transaction

Concurrent execution of user programs is essential for good DBMS performance.
A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is R&W from/to the database.
A transaction: a sequence of reads and writes operations on the objects in DB.
end in three ways:

- commit: is done and make the updates .
- system abort: system cannot perform the transaction because of something bad happen.
- user abort: the user aborts the transaction.

ACID properties for a transaction:
A (atomicity) : All actions in the Xact happen, or none happen. A transaction has to be ended with commit or abort. Cannot stuck in a intermediate state. If each Xact is consistent, and the DB starts consistent, it ends up consistent.
C (consistency): start at a consistent state, a transaction should bring the system into another consistent state. In reality, it is hard to define this, as transactions happen all the time. So serializable schedule is consistent.
I (isolation): Execution of one Xact is isolated from that of other Xacts. more than one transactions occurring at the same time. execute transaction sequentially. Throughout is a problem. if do simultaneously, problems.
D (durability): if commit, from that point on, the transaction will not be lost. we do not make all the writes to the disk. we have buffer management layer to improve efficiency. If a Xact commits, its effects persist
C and I left to the concurrency manager.
A and D left to the recovery manager.

Scheduling Transactions

Motivation for Concurrent Execution:

- Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases system throughput
- Interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly.

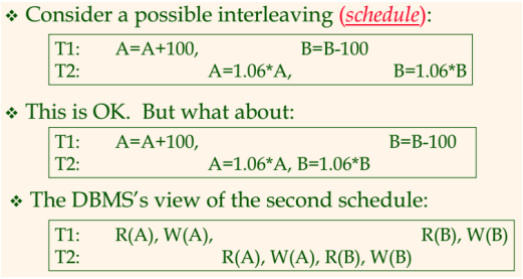
A schedule is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T.

Serial schedule: Schedule that does not interleave the actions of different transactions. Definitely consistent but no concurrency.

Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

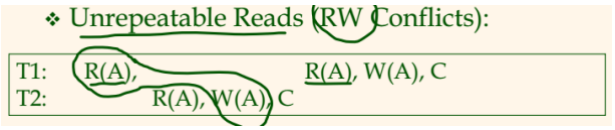
Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.

Anomalies Due to Interleaved Execution

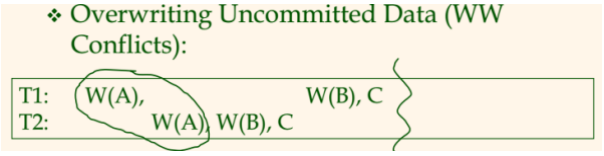


The second schedule is a WR conflict.

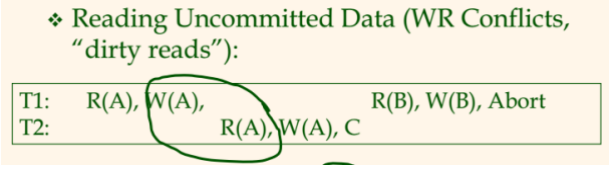
The general problem illustrated here is that T1 may write some value into A that makes the database inconsistent. interleaved execution can expose this inconsistency and lead to an inconsistent final database state. While a serial schedule will not expose this inconsistency.



Everything should be isolated, two reads in one transaction should be the same.



T1 commit latter but the value it changed had been changed by another T2 which committed earlier. Lost update. A and B should be written by one transaction.



T2 read after T1 write something, then T1 abort. But T2 commit.

We say that such a schedule is unrecoverable. In a recoverable schedule, transactions commit only after all transactions whose changes they read commit.

In this abort situation. A serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions in S.

Transactions happen at runtime. do not have too much time to figure out a concurrent

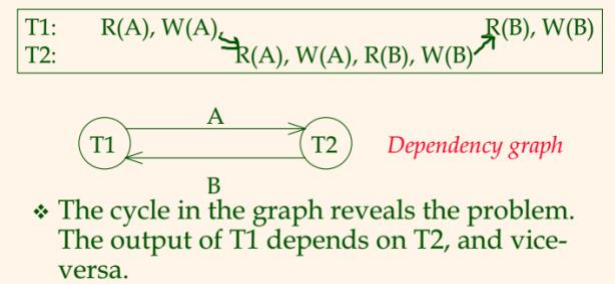
schedule which is also consistent.
More concurrency may need more time to check consistency. Simple consistency may have low concurrency.
Serializable schedules are consistent. any schedule equals a serializable schedule is a constant one.
NoSQL allow something not consistent as long as no serious problems.

Conflict Serializable Schedules

give up some serializable schedules, which are also serializable but hard to check. only focus on conflict SS subset.
Conflict does not necessarily mean problem. There is a potential for a problem.
The outcome of a schedule depends only on the order of conflicting operations; we can interchange any pair of nonconflicting operations without altering the effect of the schedule on the database. If two schedules are conflict equivalent, it is easy to see that they have the same effect on a database.
Two schedules are conflict equivalent if:

- Involve the same actions of the same transactions
- Every pair of conflicting actions is ordered the same way(that means if we draw arrows between two Xacts to show the dependency, they all point to the same direction)

Schedule S is conflict serializable if S is conflict equivalent to some serial schedule
Precedence graph: One node per Xact; edge from Ti to Tj if Tj reads/writes an object last written by Ti.
Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.



By doing cycle checking in a graph, can catch all non-conflict serializable transactions. But other schedules that may be also serializable and more concurrent will be missed.
Building and checking graph is expensive. Then we try to avoid cycles.

LOCK-BASED CONCURRENCY CONTROL(CC)

A lock is a small bookkeeping object associated with a database object.
A locking protocol is a set of rules to be followed by each transaction to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

- have to maintain locks.
- atomicity is needed in lock management.

- more lock less concurrent.
- may cause abort or blocking.

Strict 2PL

Having two types of locks: shared lock and exclusive lock. Only S locks compatible to each other.

- Before read, obtain a S lock. before write, obtain a E lock.
- keep all locks until finish(commit or abort).

One phase is obtaining locks, # of locks is growing; the other is lock steady state. To certain extend, Strict 2PL only allow serial schedules which are safely partially overlapped.
Strict 2PL guarantees that a precedence graph is acyclic. When conflicts happen, have to wait until others commit.

2PL

Can release locks early. Once release, cannot acquire locks anymore.
The two phases become growing phase and shrinking phase.
Still guarantee acyclic. But, if abort, could violate isolation. That is, before a Xact commits, it allow other Xacts to see the inconsistent states caused by it. Like the one in previous anomalies.

Dead lock

Necessary four conditions(Coffman conditions):

1. mutual exclusion: at least one process must be held in a non-sharable mode.
2. hold and wait: there must be a process holding one resource and waiting for another.
3. No preemption: resources cannot be preempted.
4. circular wait: there must exist a set of processes

How to detect dead lock:

- use waits-for graph. nodes are transactions. Having arrows indicate one is wait a lock in another transaction. Do cycle check.
- a timeout mechanism

Abortion can resolve deadlock. But abortion in PL2 can violate isolation.
How to prevent:
Timestamp base deadlock prevention. Do not cause cycle in the graph:
usually give high priority to older transaction, as they have done some work.
Assume Ti wants a lock that Tj holds. Two policies are possible:

- Wait-Die: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts(lower-priority transactions can never wait for higher priority transactions)
- Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits(higher-priority transactions never wait for lower-priority transactions)

If a transaction re-starts, make sure it has its original timestamp.

In either ease, no deadlock cycle develops.

Optimistic 2PL

- Set S locks as usual.
- Make changes to private copies of objects.
- Obtain all X locks at end of Xact, make writes global, then release all locks.

In contrast to the following Optimistic CC as in Kung-Robinson, this scheme results in Xacts being blocked, waiting for locks.
However, no validation phase, no restarts.

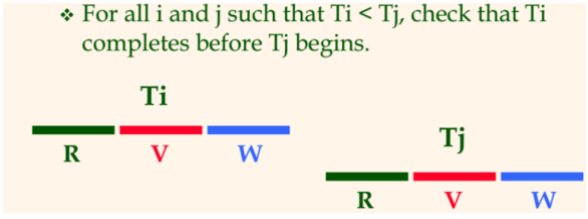
CC WITHOUT LOCKING

why not use timestamps to solve dependency instead of locks to prevent cycles. but need to care about each object, maintain more information. Whole idea is make the dependency graph only has one direction arrows.

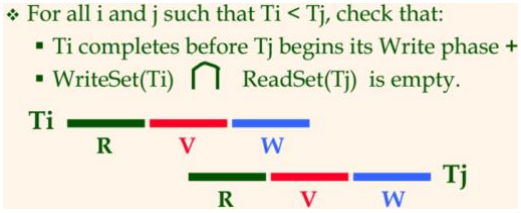
Optimistic CC(Kung-Robinson Model)

Three phases:
READ: Xacts read from the database, but make changes to private copies of objects.
VALIDATE: Check for conflicts.
WRITE: Make local copies of changes
In validation:

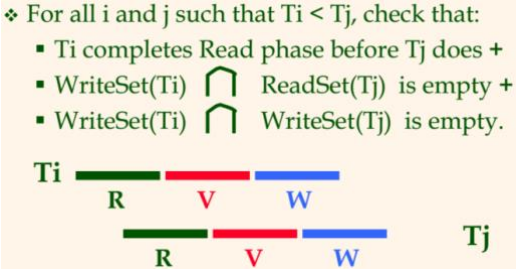
- Test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact is assigned a timestamp id, just before validation begins.
- ReadSet(Ti): Set of objects read by Xact Ti; WriteSet(Ti): Set of objects modified by Ti.
- Three test cases:



In this case, no interleaving. Tj only see what Ti committed. avoid RW conflicts



In this case, test WW(no overwrite), WR conflicts(no dirty read).



In this case, test WR and WW conflicts.

Serial validation: Put V and W phases into a critical section.

- no need to check test 3
- but if V and W are slow(large set or write remotely), major drawbacks
- Opt for read-only Xacts.

Overheads:

- need supports from OS as for critical section.
- maintain sets
- set operation could be slow
- need to restart Xacts failing validation

Timestamp CC(give consistency not isolation unless wait)

Idea: Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins.

In this case, If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj.

When Xact T wants to read Object O (No RR, only check WR)

- If TS(T) < WTS(O), WR conflicts. So, abort T and restart it with a new, larger TS.
- If TS(T) > WTS(O): Allow T to read O. Reset RTS(O) to max(RTS(O), TS(T))

When Xact T wants to Write Object O(check WW RW)

- If TS(T) < RTS(O), RW conflicts. abort and restart T.
- If TS(T) < WTS(O), WW conflicts. Thomas Write Rule: We can safely ignore such outdated writes; need not restart T! (T's write is effectively followed by another write, with no intervening reads.) Allows some serializable but non conflict serializable schedules
- Else, allow T to write O.

The above protocol is unrecoverable.

Timestamp CC can be modified to allow only recoverable schedules:

- Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
- Block readers T (where TS(T) > WTS(O)) until writer of O commits.

Multi-version Timestamp CC(no wait except for isolation)

Idea: Let writers make a "new" copy while readers use an appropriate "old" copy:

Readers are always allowed to proceed.

Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most

recently read this version as its RTS.
Versions are chained backward; we can discard versions that are “too old to be of interest”.
Each Xact is classified as Reader or Writer. Writer may write some object; Reader never will.*f*
Reader Xact:
For each object to be read: Finds newest version with WTS < TS(T).(Avoid WR conflicts)
Reader Xacts are never restarted. However, might block until writer of the appropriate version commits.
Writer Xact:
To read an object, follows reader protocol.
To write an object:
Finds the newest version V s.t. WTS < TS(T).(avoid WW)
If RTS(V) < TS(T)(avoid RW), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Xacts can see TS values but can’t read version CV.)
Else, restart with a higher TS.

Reading a non-fleshing objects, but keep the system consistent.
maintain multiple versions
isolation still a problem.