Reference Sheet for CO130 Databases

Spring 2017

1 Benefits of Databases

1. Organised and efficient.
2. Minimise data duplication.
3. Support concurrent actions.
4. Support multiple users, controls who can access which data.
5. Support recovery from failures.

1.1 Transactions

1. *Atomicity*: if one part fails, whole transaction fails (rolls back).
2. *Consistency*: must not leave database in inconsistent state.
3. *Isolation*: executed as if no other transaction is occurring (serialised execution).
4. *Durability*: Results of successful transactions not lost.

2 Relational Model

1. *Database*: one or more relations.
2. *Database schema*: schemas for all relations.
3. *Relation*: heading and body: set of tuples of the form $(A_1 : S_1 = V_1, \ldots, A_n : S_n = V_n)$.
4. *Relation schema*: name of relation and heading.
5. *Heading*: unordered set of attributes (names and types).
6. *Body*: unnordered set of tuples (sets of attribute values).

2.1 Entity-Relationship Digrams

1. *Entity sets* (rectangles): distinguishable entities tht share same properties.
2. *Relationship* (diamonds): captures how two or more entity sets are related.
3. *Attributes* (circles): property of entity, primary key attributes underlined.

2.1.1 Complex Attributes

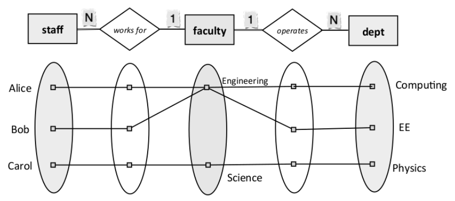
1. *Composite* (tree): subdivided e.g. address into road, city, postcode.
2. *Multivalued* (double border): set of values e.g. several phones.
3. *Derived* (dotted line and border): computed from other values e.g. age from DoB.

2.1.2 Cardinality Constraints and Entity Set Participation

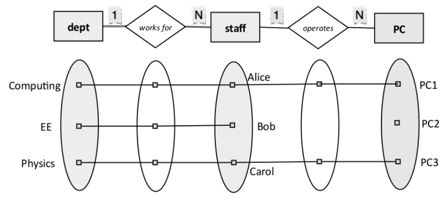
1. *One-to-one*, *one-to-many*, *many-to-many*: add labels 1 1, 1 N, and M N to relationship respectively.
2. *Total participation*: use double line, requires all entities to participate in a relationship.
3. *Participation bounds*: use .. notation in labels.

2.1.3 Fan Traps and Chasm Traps

Fan traps: ambiguous paths exist between entities. Can be solved by changing structure (e.g. staff - dept - faculty).



Chasm traps: suggests a relationship between entities but one doesn't exist. Can be solved by adding a new relationship (e.g. between dept and PC).



2.1.4 More relationships

1. *Multiway relationships*: can have relationships between more than two entity sets.
2. *Roles*: one entity set can have more than one role (e.g. prequel and sequel), draw multiple lines and label.
3. *is-a relationships*: hollow arrow head, can form hierarchies.

2.1.5 More Entities

Weak entities (double rectangle, double diamond): cannot be uniquely identified on own attributes and requires a strong entity to exist. Identified using primary key of strong entity and one ore more attributes of itself (dotted underlined). E.g. room cannot exist without building that contains it.

2.2 Relation Schemas from ER Models

2.2.1 Joins and Keys

1. Primary key: uniquely identifies tuple
2. Foreign key: primary key used in different table

2.2.2 Entity Sets and attributes

Mapped directly to realtion with the same attributes. E.g.

```
actor(ID, firstname, lastname, housno, roadname, city)
actor_cars(actorID, carID)

create table actor (
  ID            int,
  firstname     varchar(30),
  lastname      varchar(30),
  houseno       int,
  roadname      varchar(30),
  city          varchar(40),

  primary key (ID)
)

create table actor_cars (
  actorID       int,
  carID         varchar(10),

  primary key (actorID, carID),
  foreign key (actorID) references actor.ID
)
```

1. *Composite attributes*: flatten to contain only simple attributes.
2. *Multivalued attributes*: own relation mapped back to entity set using foreign key.
3. *Derived attributes*: not supported by relational model.

Weak Entity Sets Mapped to its own relation and attributes but includes primary key of strong attribute with on delete cascade constraint.

2.2.3 Relationships

Many-to-many: new relation with two foreign keys. E.g.

```
person(ID, otherattributes)
car(regno, otherattributes)
drive(personID, regno, otherattributes)

create table drive (
  personID    varchar(10),
  regno       varchar(12),

  primary key (personID, regno),
  foreign key (personID) references person.ID on delete.cascade,
  foreign key (regno) references car.regno on delete.cascade
)
```

One-to-many and *one-to-one*: directly include primary key of the One relation as foreign key in the Many relation. E.g.

```
person(ID, otherattributes)
car(regno, personID, otherattributes)
```

```
create table car (
  regno       varchar(12),
  personID    varchar(10),

  primary key (regno),
  foreign key (personID) references person.ID
)
```

Multiway Relationships Include primary keys from all entity sets of foreign keys. Primary key is formed by foreign keys of the many entity sets.

Roles in Relationships Each role is mapped to a foreign key attribute.

is-a Relationships Include primary key of root level entity set with every lower-level entity set

2.2.4 Database Design Maxims

1. Model the 'real world' as much as possible
2. Keep it as simple as possible
3. Express each property once only

3 Relational Algebra

1. *Union* $R \cup S$: set of tuples in R or S.
2. *Difference* $R - S$: set of tuples in R but not S.
3. *Intersection* $R \cap S$: set of tuples in R and S.
4. *Projection* $\pi_A(R)$: relation R with attributes A only.
5. *Condition* $\sigma_p(R)$: all tuples of R that satisfy the condition p.
6. *Cartesian product* $R \times S$: all tuples that can be paired by including a tuple from R and a tuple from S.
7. *Natural join* $R \bowtie S$: all tuples that can be 'joined' using all matching attributes of R and S.
8. *Left outer*, *right outer*, *full outer join*: keep entries from left, right, both set(s) which do not match attributes in the other.
9. *Renaming* $\rho_{n_1/o_1,\ldots,n_n,o_n}(R)$: changes name of attribute o_1 to n_1, ..., o_n to n_n.

Relational Expressions Used for *retrieval*, *updates* (insert, change, delete data), defining *constraints*, *derived relations* (define one relation in terms of others), *concurrency* (defining data to use for concurrent actions), *access control* (define data over which permissions should be granted).

4 Functional Dependencies

1. *Functional Dependency*: constraint that if two tuples of a relation R agree on attributes $A_1, A_2, \ldots, A_n$ they also agree on $B_1, B_2, \ldots, B_m$: $A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$.
2. *Superkey*: a set of attributes that functionally determines all the other attributes of the relation.
3. *Candidate Key*: if there is no proper subset of the superkey.

Splitting and Combining Dependencies

1. *Splitting Rule*: if $A, B, C, D, E \rightarrow X, Y, Z$ then $A, B, C, D, E \rightarrow X$, $A, B, C, D, E \rightarrow Y$ and $A, B, C, D, E \rightarrow Z$.
2. *Combining Rule*: if $A, B, C, D, E \rightarrow X$, $A, B, C, D, E \rightarrow Y$ and $A, B, C, D, E \rightarrow Z$ then $A, B, C, D, E \rightarrow X, Y, Z$.

Trivial Dependencies

1. *Trivial Dependency Rule*: if $A, B, C, D, E \rightarrow A, B, X, Y, Z$ then $A, B, C, D, E \rightarrow X, Y, Z$ and vice versa.
2. *Trivial FD*: all attributes on the RHS are also on the LHS.

Closure of Attribute Sets

1. Consider a set of attributes L. Its *closure under* F, L^+ is the set of all attributes functionally determined by L.
2. L is a superkey of R if L^+ contains all the attributes of R.
3. If RHS $\subseteq$ LHS$^+$, then LHS $\rightarrow$ RHS.

Armstrong's Axioms Sound and complete axiomatisation of FDs.

1. *Reflexivity*: $\alpha \rightarrow \beta$ always holds if $\beta \subseteq \alpha$
2. *Augmentation*: if $\alpha \rightarrow \beta$ then $\alpha\gamma \rightarrow \beta\gamma$ (note that $\alpha\gamma = \gamma\alpha$).
3. *Transivitiy*: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$.

Additional Rules Can be derived from the axioms.

1. *Union*: if $\alpha \to \beta$ and $\alpha \to \gamma$ then $\alpha \to \beta\gamma$.
2. *Decomposition*: if $\alpha \to \beta\gamma$ then $\alpha \to \beta$ and $\alpha \to \gamma$.
3. *Pseudotransitivity*: if $\alpha \to \beta$ and $\delta\beta \to \gamma$ then $\delta\alpha \to \gamma$.

Finding Closure of Functional Dependency Set Start with $F^+ = F$. Repeat (until F^+ doesn't change):

1. Apply reflexivity and augmentation. Add new FDs to F^+.
2. Apply transivity to FDs in $F^+$and add new FD to F^+.

Covers of Functional Dependency Sets

1. F_1 and F_2 are equivalent if each implies the other. They are *covers* of each other.
2. A cover is *canonical* if:
   (a) Each LHS is unique.
   (b) We cannot delete any FD from the cover and have an equivalent FD set.
   (c) We cannot delete any attribute from any FD and still have an equivalent FD set.

Computing a Canonical Cover Repeat over F (until it doesn't change):

1. Union rule for all possible dependencies (if $\alpha \to \beta$ and $\alpha \to \gamma$ then $\alpha \to \beta\gamma$).
2. Remove any extraneous attributes (often easiest by inspection).
   (a) LHS X is extraneous if $\text{RHS} \subseteq \{\text{LHS} - X\}^+$ under the FD set.
   (b) RHS X is extraneous if $X \in \text{LHS}^+$ under the FD set with X removed from its FD RHS.

5 Normalisation

5.1 Decomposition

1. *Decomposition*: Given R, decompose it into S and T such that $\text{attr}(R) = \text{attr}(S) \cup \text{attr}(T)$, $S = \pi_{\text{attr}(S)}(R)$, $T = \pi_{\text{attr}(T)}(R)$.
2. *Lossless Decomposition*: At least one of the following FDs holds:
   (a) $\text{attr}(S) \cap \text{attr}(T) \to \text{attr}(S)$
   (b) $\text{attr}(S) \cap \text{attr}(T) \to \text{attr}(T)$
3. *Dependency Preserving Decompositon*: We can check the functional dependencies of R without joining S and T.

5.2 Boyce-Codd Normal Form

A relation is in *BCNF* iff for all non-trivial FDs, the LHS of every FD is a superkey (contains a key).

Decomposition into BCNF For a relation R. While there are relations V with violating FDs:

1. Find a non-trivial FD LHS $\to$ RHS s.t.:
   (a) LHS is not a superkey of R
   (b) $\text{LHS} \cap \text{RHS} = \{\}$
2. Remove V from the current decompositions
3. Add relation $(\text{LHS} \cup \text{RHS})$ to decompositions.
4. Add relation $(\text{attr}(V) - \text{RHS})$ to decompositions.

Note that BCNF is lossless but not necessarily dependency preserving. It eliminates redundancy.

5.3 Third Normal Form

A relation is in *3NF* iff for all non-trivial FDs, the LHS of every FD is a superkey or if every attribute on the RHS of a FD is prime (a member of any key of the relation).

Decomposition into 3NF C is a canonical cover (minimal FD set) for R. Then start with the set $D = \{\}$ of decomposed relations.

1. For each FD LHS $\to$ RHS in C, add a new relation $(\text{LHS} \cup \text{RHS})$ to the set of decomposed relations D.
2. For each relation R in D that is a subset of another relation in D, remove R from D.
3. If none of the relations in D includes a key for R, add a new relation(key) to D.

3NF is lossless and dependency preserving but does not necessarily eliminate redundancy.

6 Structured Query Language

6.1 Relational Algebra to SQL

Relation Algebra	*SQL*
$R \cup S$	`R union S`
$R \cap S$	`R intersect S`
$R - S$	`R except S`
$\pi_{\text{attributes}}(R)$	`select attributes from R`
$\sigma_{\text{condition}}(R)$	`from R where condition`
$R \times S$	`R,S or R cross join S`
$R \bowtie S$	`R natural join S`
$R \bowtie_{\text{condition}} S$	`R join S on condition`
Relation	`Table`
Relational Expression	`Views`
Tuple	`Row`
Attribute	`Column`
Domain	`Type`

6.2 Some Common Types

1. `int, smallint, real, double precision, float(n), numeric(p,d), decimal(p,d)` - usual arithmetic operators are available.
2. `char, char(n), varchar(n), clob/text, ...` - operators include || (concatenation), `like` (performs pattern matching: _ for any char, % for zero or more chars), `similar to` (for regex matches).
3. `bit(n), byte(n), blob`.
4. `boolean` - based on three-valued logic (can be `unknown`) - operators include `between`, `not between`, `in`, `not in`.
5. `date, time, timestamp`.
6. `...`

6.3 Null

Attribute value used to represent values that are missing, not applicable, or witheld.

1. Any arithmetic involving `null` results in `null`.
2. Comparisons with `null` give `unknown`.
3. `null` will never match another value, need to use `is null` or `is not null`.

6.4 Queries

1. `select atts from table where conds` used to query datbase.
2. `*` used to select all attributes.
3. `as newName` used to rename attributes.
4. `order by att` (`asc` or `desc`) used to sort results by a column.
5. `table inner join table`, `left outer join`, `right outer join`, `full outer join`.
6. `on cond` to join given a predicate.
7. `using att` to join on a specific attribute.
8. `distinct` to eliminate duplicates.
9. `sum`, `avg`, `min`, `max`, `count` as aggregate functions.
10. `group by` ... `having` used to group tuples in resulting relation. Note any non-aggregates used in the `having` filter must be included in the `group by` list.
11. `some` (any) and `all` to compare a value against some or all values returned by a subquery.
12. `exists` and `not exists` to test whether a relation is empty or not
13. `unique` or `not unique` to test if a relation has duplicates.

6.4.1 Subqueries

1. *Scalar subquery* produces single value. Typically a `select` with an aggregate function.
2. *Set subquery* produces set of distinct values (column). Typically used for membership (`in` or `not in`) or comparisons (`some` or `all`).
3. *Relation subquery* produces relation. Typically used as operand of products, `joins`, `unions`, `intersects`, `excepts`, `exists`, `not exists`, `unique` or `not unique`.

6.4.2 Advanced Queries

1. `limit n` to specify number of records to return.
2. `like ...` used to search for specified pattern, _ matches exactly one character, `%` matches zero or more characters.

6.5 Data Definition

1. `create table name (atts)` to create a relation.
2. `drop table table` to delete a relation.
3. `alter table table`, then `add (atts)` or `drop (atts)`.

Constraints

1. `not null` prohibits assignment of `nulls`.
2. `default` sets default value for attribute.
3. `auto_increment (start, by)` allows a unique number to be generated by default.
4. `primary key (atts)` (must be unique, `nulls` not permitted).
5. `foreign key (atts) references table (atts)` checks that attributes in relation match value of a candidate key in a referenced relation (*referential integrity*).
6. `unique (atts)` to ensure no two tuples have same set of values for listed attributes.
7. `check (expr)` can check an arbitrary expression involving several attributes and/or a query.
8. Constraints should be named using `constraint constraintName ...`.
9. `on update cascade`, `on delete cascade` maintains referential integrity by cascading updates / deletions to the foreign key.
10. `on update set default`, `on delete set null` another option that will lead to unmatched tuples.

6.5.1 Views

Relations defined using a query, not physically stored. Can be used to:

1. Declare commonly used subqueries.
2. Declare a relation over several relations using products, joins.
3. Declare a relation over calculated expressions and aggregated data.
4. Partition data using a selection.
5. Restrict access to a relation by providing access only to a view.

Materialised views, stored and kept up to date rather than being recomputed each time. *Updatable views* usually don't make sense.

6.5.2 Indexes

Copies of an attribute's data that are automatically maintained by RDBMS but can be searched quickly. Use `create index name on table (atts)`.

6.6 Data Manipulation

1. `insert into table (atts) values (atts), (atts), ...` adds tuples to a relation. Can use a subquery instead of inserting individual tuples.
2. `delete from table where ...` to delete tuple attributes.
3. `update table set ... where ...` updates tuple attributes.
4. `case when ... then else end` is often useful in update expressions.

7 Transactions

1. `start transaction` explicitly starts a transaction.
2. `commit` and `rollback` commit and rollback modifications respectively.

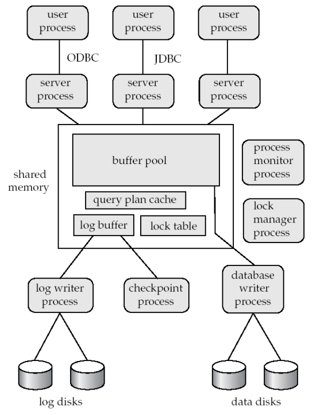
Ensuring ACID:

1. *Atomicity and Durability*: Rollback possible using a recovery system. DBMS checks logs. Compensating transactions must be possible. RAID / replicated disks should be used to guard against disk failure.
2. *Consistency*: Up to the programmer to ensure using constraints.
3. *Isolation*: Use a concurrency control system. E.g. allow concurrent reads or concurrent read / writes if they are on different relations. Isolation levels:
   (a) *Read uncommitted*: uncommitted data changed by other concurrent transactions can be read - *dirty reads*.
   (b) *Read committed*: Only data *committed* by other concurrent transactions can be read. *Non-repeatable reads* are possible.
   (c) *Repeatable read*: Guarantees all tuples can be returned by a query will be included if the query is repeated, but might return additional newly committed tuples (*phantom reads*).
   (d) *Serializable*: Guarantees isolution.

8 Storage and File Structure

8.1 Database Servers

1. Clients send requests, transactions executed on server, results shipped back to client.
2. Communicated through a remote procedure call.
3. Open Database Connectivity / JDBC are API standards for sending SQL requests.



All database processes can access shared memory. Systems implement mutual exclusion using semaphores or atomic instructions.

8.2 Storage

1. *Volatile storage*: loses contents when power switched off.
2. *Non-volatile storage*: contents persist even when power switched off.

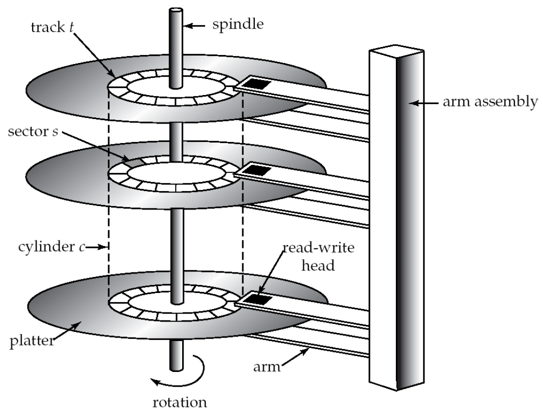
Physical storage media

1. *Cache*: fastest and most costly; volatile; managed by system hardware.
2. *Main memory*: fast; too small or expensive to store a database; volatile.
3. *Flash memory*: fast reads but slow writes; can only written to once - but locations can be erased and written to again (limited number of write/erase cycles, erasing has to be done to entire bank of memory); non-volatile.
   (a) NOR: lower capacity, sometimes used to store program code in embedded devices.
   (b) NAND: higher capacity, sometimes used as portable data storage.
4. *Magnetic disk*: uses spinning disk, magnetic reads and writes; long-term storage of data; data must be moved through main memory; direct-access (read in any order); survives power-failure and system crashes (but rare disk failures).
5. *Optical storage* (e.g. CD-ROM, DVD): slow reads and writes; uses spinning disk and laser; non-volatile; used in jukebox systems.
6. *Tapes*: slow; high capacity; sequential-access; non-volatile; used for backups and archives - tape jukeboxes can store hundreds of terabytes. Cheapest storage medium.

Storage Hierarchy

1. *Primary storage*: Fastest media but volatile (cache, main memory).
2. *Secondary storage*: Non-volatile, moderately fast, on-line storage (fash, magnetic disks).
3. *Tertiary storage*: Non-volatile, slow access time (magnetic tape, optical storage).

Magnetic Disks



Disk controller: interfaces between system and disk drive.

1. Accepts commands to read or write to sector.
2. Moves disk arm to right track and reads / writes.
3. Computes and attaches checksums to each sector to verify data is read back correctly.
4. Ensures successful writing by reading back sector after writing.
5. Performs remapping of bad sectors.

Measuring performance:

1. *Access time*: time for a read or write request to be issued.
   (a) *Seek time*: time to reposition arm over correct track.
   (b) *Rotational latency*: time taken for sector to be accessed.
2. *Data transfer rate*: rate at which data can be retrieved from or stored to disk.

Optimisations for block access:

1. *Block*: contiguous sequence of sectors from a single track. Units of storage allocation / data transfer.
2. *Disk arm scheduling algorithms*: order pending accesses to tracks to minimise disk arm movements.
   (a) E.g. *elevator algorithm*: move disk arm in one direction until no more requests in that direction, then reverse.
3. *File organisation*: Organise blocks to correspond how data is accessed.
   (a) Store related information on same blocks / cylinders.
   (b) Files can get *fragmented* over time (by additions / scattered free blocks). Defragment utilities often available.
4. *Non-volatile write buffers*: write blocks to non-volatile RAM (battery backed RAM or flash memory) buffer immediately.
   (a) Controller writes to disk when no other requests or if request pending for long time.
   (b) Writes can be reordered to minimise disk arm movement.
5. *Log disk*: write a sequential log of block updates. Used like non-volatile RAM. Fast since no seeks required.
6. Reorder writes to improve performance.
   (a) *Journaling file systems*: write data in safe order to NV-RAM or log disk. Otherwise risk of corruption of file system data.

Storage Access

1. Minimise block transfers by keeping as many blocks as possible in main memory.
2. *Buffer*: portion of main memory available to store copies of disk blocks.
3. *Buffer manager*: subsystem responsible for allocating buffer space.
   (a) If block already in buffer, returns memory addr.
   (b) Otherwise allocates space in buffer for the block, replacing some other (usually least recently used) block if required. Replaced block written back to disk only if it's been modified.
   (c) Read block from disk to buffer and return addr of block.
4. Buffer replacement policies can be least-recently used (LRU) or more complex:
   (a) *Pinned block*: memory block to not be written back to disk.
   (b) *Toss-immediate*: frees space as soon as has been processed.
   (c) *MRU*: pins current block. After final tuple processed, unpins block, and becomes MRU.
   (d) *Statistical*.

File Organisation Database is a collection of *files*, which is a sequence of *records*.

1. *Fixed length records*: Assume record size fixed, only holds one particular type, different files for different relations. Deletion - either shift all records, or free list of deleted records with links.
2. *Variable length records*: e.g. *Slotted page*: header contains number of entries, end of free space, location and size of each record. Records can be moved around. Pointers should point to header entry.

Sequential file organisation:

1. For sequential processing, ordered by a *search key*.
2. *Deletion*: use pointer chains.
3. *Insertion*: update pointer chain - if free space insert there, otherwise in an overflow block.
4. Need to reorganise file sometimes to maintain sequential order.

Organisation of records in files:

1. Heap: record can be placed anywhere in file.
2. Sequential: store records in sequential order, based on value of search key.
3. Hashing: hash function on some attribute specifies in which block record is placed.
4. Multitable clustering file organisation: records of several relations stored in same file. Related records can be stored on the same block.

Can be *row-oriented* (good for selecting all attributes) or *column-oriented* (good for selecting one attribute). Can be sorted to optimise queries like `where att` > `val`.

9 Indexing

1. An *index file* consist of records (*index entries*) of the form (*search-key*, pointer).
   (a) *Ordered indices*: search keys in sorted order. For a sequentially ordered file, can use a primary index (clustering index) or secondary index.
   (b) *Hash indices*: distributed accross buckets using hash function.
   (c) *Dense index files*: Index record for every search-key value.
   (d) *Sparse*: Index records only for some search key values.
      i. Usually requires less space and maintenance, but slower.
      ii. *Tradeoff*: sparse index with index entry for every block in file.
   (e) *Multilevel index*: treat primary index as sequential file and create a sparse index for it. Keep outer index small enough to fit in main memory.

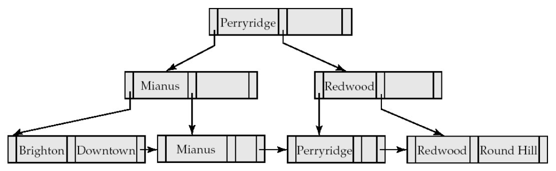
Keeping indices up to date:

1. *Deletion*: delete search key, or for sparse index, replace by next search key value.
2. *Insertion*: insert search key if not present, or for sparse index, only if new block created.

Secondary Indices:

1. Points to a bucket that contains pointers to all records with that search-key value.
2. Much more expensive to scan.

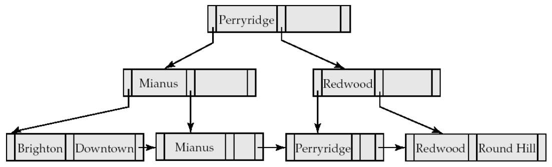
B^+ Tree Index Files



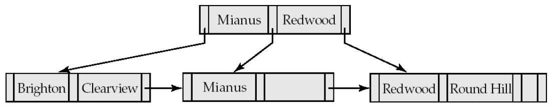
Performance of indexed-sequential files degrades over time, due to overflow blocks. B^+ trees are a better solution:

1. All paths from root to leaf are the same length.
2. Each node that is not a root or leaf has between $\lceil n/2 \rceil$ and n children.
3. A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values.
4. *Node structure*: $(P_0, K_1, P_1, \ldots, K_n, P_n)$, where K are search key values, P are pointers to children / records / buckets of records.

Insertion: Insert if room on leaf node, otherwise split node and propogate upwards.



Deletion: Remove from leaf, if too few records then merge siblings or redistribute. If node has only one pointer, then propogate upwards.



Indexing of strings often uses prefix compression.

9.1 Hashing

1. *Bucket* contains one or more records.
2. Obtain bucket of record using hash function.
3. Records with different search-keys may end up in the same bucket.

Ideal hash functions are *uniform* and *random*.