COSC265 — Relational Database Systems

Neville Churcher

Department of Computer Science & Software Engineering
University of Canterbury

2021

Index Structures

- ☆ Physical order of records on disc affects efficiency of individual queries requiring file traversal.
- ☆ `select * from node where label = 'root'`
- ☆ Physical records can only be in one order at a time!
- ☆ Need an efficient way to access records in different orders without expensive operations like physical sorting
- ☆ Index maps *values* of indexing field/attribute to corresponding logical *records*
- ☆ Index entries ordered by indexing field value
- ☆ Typically implemented as indexing field values with list of pointers to disc blocks containing logical records
- ☆ May have multiple indices on single file

What to Index?

Make queries go faster . . .

Primary key: to give natural sorted order

Foreign key: to assist in join operations

```
select name from node, edge
  where edge.from = node.id
```

Common access paths:

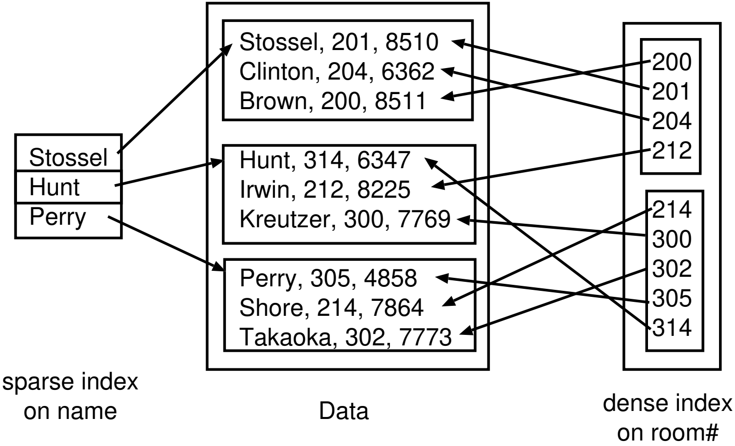
- ☆ determined at design time or
- ☆ observed in use (tuning)
- ☆ may involve attibutes that aren't in PKs or FKs

```
select * from student where gpa < 1
```

Classifying Indices

Dense: at least one index entry for each search key value

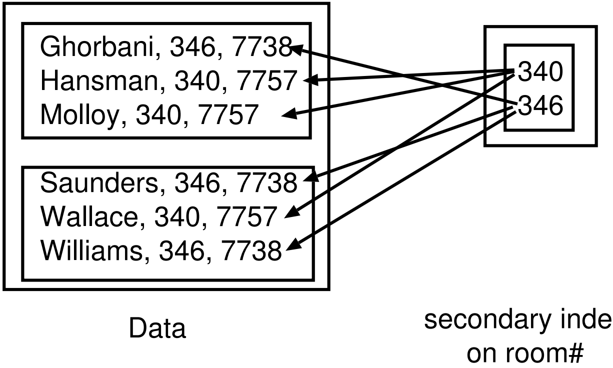
Sparse: index entries for only some search key values (typically anchor records for primary index)



Classifying Indices

Primary: includes primary key field(s). Physical order determined by unique, non-null, *ordering key field*

Secondary: may contain duplicates. Implementation may use block or record pointers



Primary Index

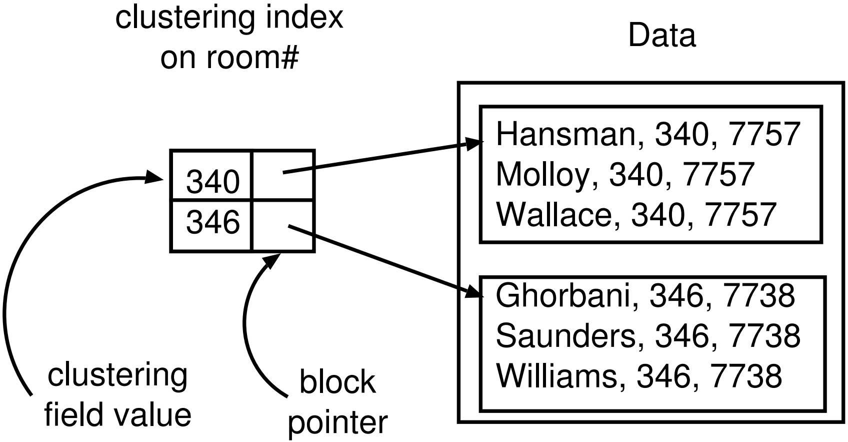
- (Ideally) Maintains entity integrity constraint
- Spot the difference...

```
create table bar (k int, primary key(k))

create table bar (k int not null)

create table bar (k int)
create unique index ki on bar(k)
```

Clustering

- Primary index specified on ordering key field.
- If ordering field is not a key (i.e. can have duplicates) then clustering index can be used
- clustering — *logically related* items placed *physically close* on disc
- Since a file can have at most one ordering field, can have at most one primary index or one clustering index *but not both*
- Clustering index is sparse—one entry per value rather than per record

Clustering Example



For really big data sets

- Index entries ordered by indexing field value
- Index files can become very large (though usually smaller than data file)
- Index traversed/searched often (otherwise pointless!)
- Single-level uses index binary search ($\log_2 N$)
- Multilevel indexing involves replacing simple (single-level) index with more complex data structure
- Time-space trade-offs
- Evolution factors—index entries may change $\not\propto$ data record updates

Index Implementation

Single-level: uses index binary search ($\mathcal{O}(\log_2 N)$)

Tree structures: speed up searching

- ☆ Internal nodes contain many pointers to other nodes
- ☆ Leaf nodes contain key values and pointers to physical data
- ☆ Balanced trees preferable when index changes often. Node split if contains less than $M/2$ keys
- ☆ Common examples include B-trees, B+-trees (no internal nodes have data pointers) and variants
- ☆ Searching order M B-tree $\mathcal{O}(\log_{M/2} N)$ — effectively constant if M large enough

Multiple Attribute Indexing

- ☆ Sometimes want to index on combination of attributes (e.g. Surname + GivenName)
- ☆ Could use single index and select from results
- ☆ Often major and minor component
- ☆ May be better to create *surrogate* attribute for combination if need more than two attributes in index

Indexing in Oracle

- Btree is default
- Some other types available (see docs)
- If the `create table` DDL statement has a `primary key` clause then Oracle will automatically create an index for the PK attributes
- However, Oracle — unlike many other DBMS — will *not* automatically create an index if there is a `foreign key` clause
- `create index by_name on customer(surname)`
- `create index by_location on supplier(country, city)`

Index Maintenance

- ☆ Costs associated with updates to index as well as data
- ☆ Insertion & deletion require moving other records as well as updating index entries
- ☆ Insertion/Deletion for primary index complicated by changes to anchor records
- ☆ DBMS implementations utilise many variations
- ☆ For bulk loading etc. it ma be more efficient to drop indexes, load data then rebuild indexes

Design Issues

- ☆ Tradeoff time/space/performance
- ☆ Tuning data (statistics, histogram, . . .)
- ☆ Query optimisation
- ☆ Predictability of queries (PK/FK, . . .)
- ☆ DBMS may silently create its own indexes for us
- ☆ DBMS may have special purpose index types (e.g. quadtrees for spatial indexing), inverted indexes for full text searching, . . .