Stored Routine

Stored routines

- a block of code that save in database call when want
- benefit: reuse code, security
- **Stored functions**
  - return something
  - call by their name not from CALL, be part of query
- **Stored procedure**
  - don't return value
  - call by CALL
- **Triggers**
  - automaticly run when something happen(insert, update, delete)
  - 2 level (ex.100 row but in 1 statement)
    - Row-level - call 100 times
    - Statement-level - call 1 time
  - create function(that return trigger) first, then create trigger (binding)
  - we can DROP, or ALTER

View - Virtual table

- don't make things faster but, easier to call
- reuse query
- update table -> view updated
- usually be read-only
- high query cost, wanna reduce overhead -> materialize view(use more storage)

EXPLAIN, EXPLAIN ANALYZE

- EXPLAIN shows how a SQL query will be executed (the query plan).
- EXPLAIN ANALYZE actually runs the query and shows the real execution steps and time.

Transaction

- update data on many places -> atomicity
- commit things ._.

Windows Function

- ประมวลชุดรายการข้อมูล โดยไม่มีการรวมรายการเหมือน aggregate function

NoSQL

- in this course we will learn Document database which is **MongoDB**

- it is **schema-less** NoSQL document database, Scale well both data volumn and network traffic
- termminology
    - table -> collection, row -> document, column -> field, table joins -> $lookup
- Document schema - JSON object that define structure in document
- pros and cons **Embedding inside document**
    - **Pros**
        - pull everything in one query
        - link data between collection through &lookup -> decrease overhead
        - ensure atomic
    - **Cons**
        - document may be too big -> overhead
        - MongoDB max at 16MB
- pros and cons **Referencing**
    - **Pros**
        - document size not too big
        - less redundency (not a big deal, if perf ok is ok)
    - **Cons**
        - use more queries or $lookup

schema design rules

1. use embedding first if possible,and one-to-few case too (one-to-many or call separately better use referencing)
2. use case that need to call separately, make it new collection
3. don't use lookup if not necessary
4. one-to-squillions(1000)(ex.log) -> new document and point back to host
5. many-to-many -> new document, point back and forth

Storage and Indexing

Overview

- how does DBMS store and access persistent data?
- minimizing I/O cost
- hash, tree based index?

External Storage

- Disks
    - Random acess devices
    - retrieve any page at fixed cost
- Tapes
    - Sequential acess device

Files and access methods layer

- Operation support (insert, delete, scan)

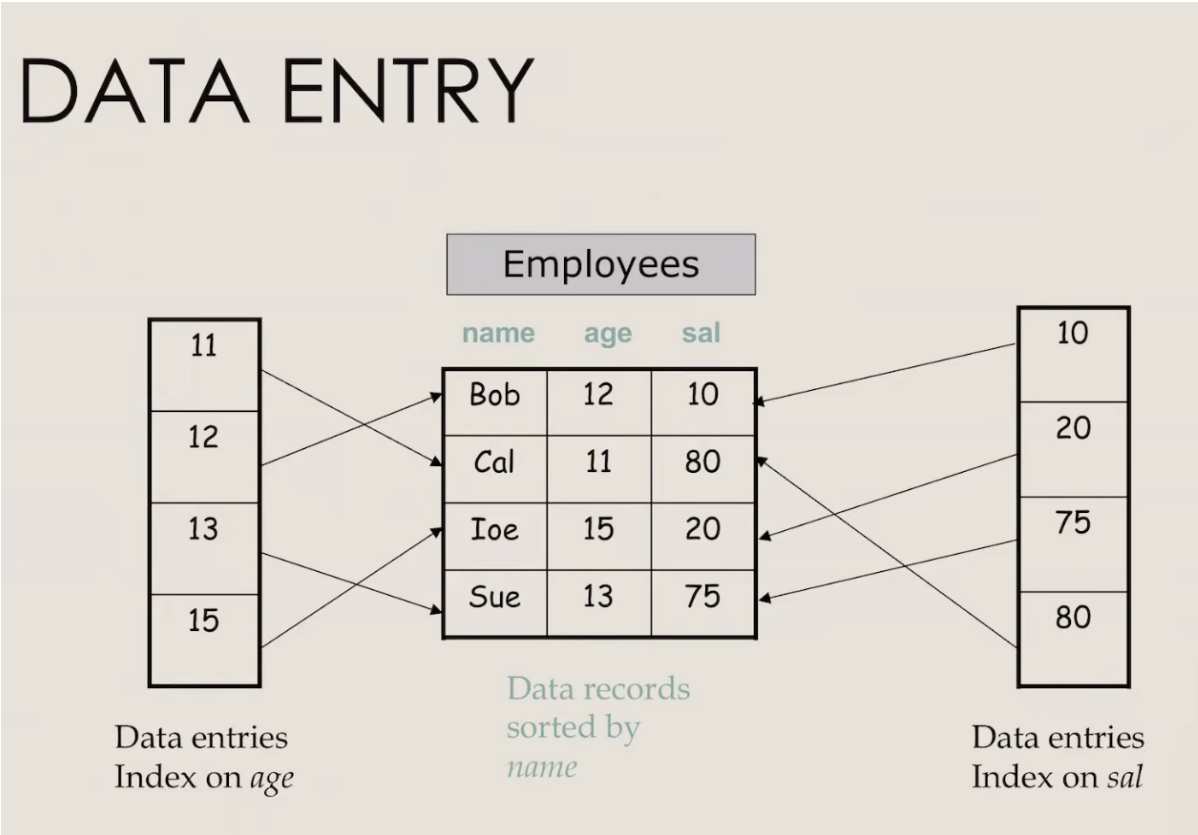
- keep track of pages allocation
- tracks available space
- how is a relation stored?
  - as a file of records, each has a **record id** which use to locate page number
  - iprementby the componect: Files and access methods layer

File Organization

- Alternative file organizations
  - heap (random order)
  - sorted
  - indexes

Concepts

- **index file** contains a collection of **data entries**(aka. index entries) -> (search-key, pointer)
- Two basic kinds of indices
  - **Ordered indices**
  - **Hash indices**



Alternative For Data Entries

- **Alternative 1**
  - data entry k* is an **actual data record**
  - at most 1 index
  - if data is large, # of pages is high
  - indexed file organization
- **Alternative 2**
  - data entry is <k,rid> pair

- **Alternative 3**
  - Data entry is <k,rid-list>
- 2,3 smaller than 1

Index classification

- **Primary index** - search key contains primary key
  - Sequential scan is efficient
- **Secondary index** - not primary._.
  - must be dense
  - Sequential scan is expensive
- (unique index - search ket contains candidate key)
- primary, secondary when file modified, every index must be update

---

- **Clustered index** - order of data recode is close to order of data entries
- **Unclustered index** - not clustered ._.

- **Dense Index Files** - index record appears for every search key value
  - faster
- **Sparse Index Files**
  - only for **Cluster and Primary** index
  - less space ans leass maintenance overhead (insert, delete)

Index Data Structure

- **HASH-BASE INDEXS**
  - index is a collection of buckets
  - good for equality selection
- **B+ TREE INDEX**
  - 50-100 fanout -> height really short
  - good for range search
  - always balance in height
  - fill factor - min pointer for a node
  - todo: understand how it balance

Cost Model

In this course we ignore CPU cost, for simplicity (I/O cost dominate), ignore pre-fetching too, use Average-case analysis

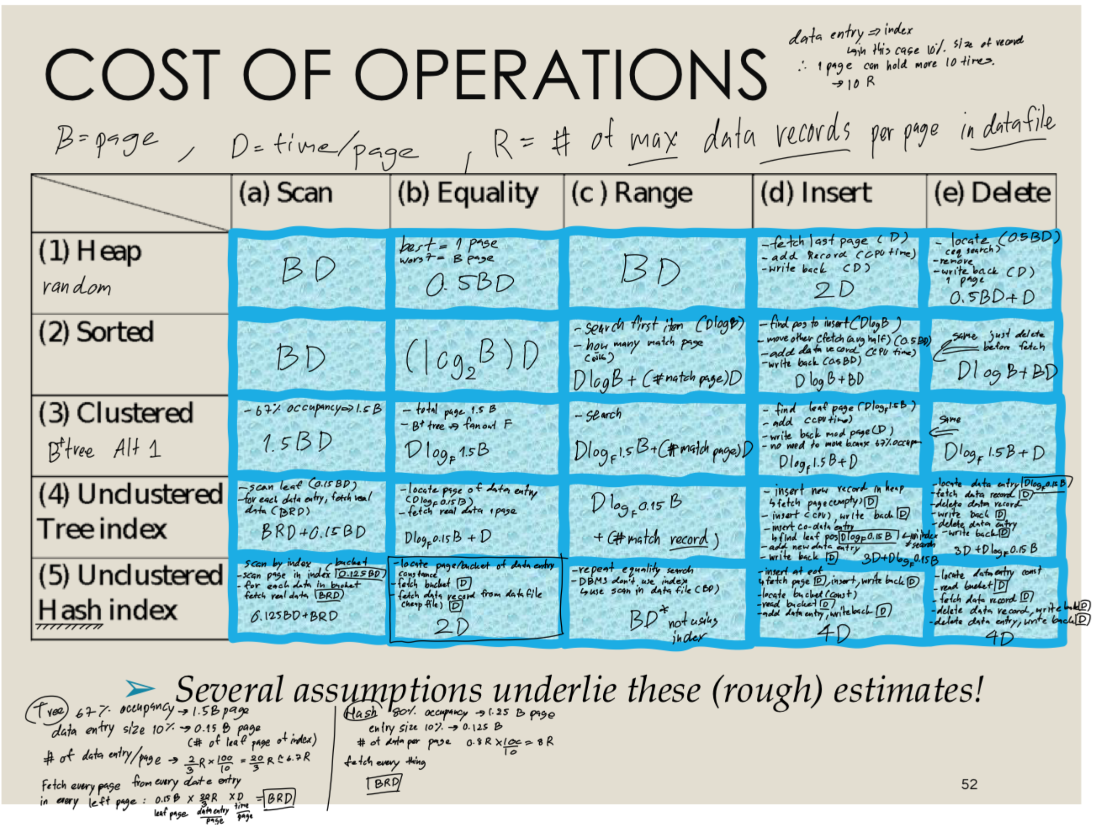
- B: The number of data page

- R: Number of data records per page

- D: AVG time to read and write disk page

- **Operations to Compare** - Scan, Equality search, Range selection, Insert selection, Insert a record, Delete a record

- **Compare File Organizations**
    - Heap files(randon order; insert at eof)
    - Sorted files, sored on a search key (no index)
    - Clustered B+ tree file on a search key, Alternative 1
    - Heap file with unclustered B+ tree index Assumtions in our analysis
- Heap files
    - equality selection on key
    - exactly one match
- Sorted files
    - files compact after deletions
- Indexes
    - Alt 2 & 3
        - data entry size = 10% size of record
    - Hash
        - No overflow bucket
        - 80% page occupany -> file size = 1.25 data size
    - Tree
        - 67% occupancy (typical) -> file size = 1.5 data size



- Understanding the Workload
    - more selective, you better index it
- Choice of indexes

- consider the most important queries in turn
- trade-off: queries faster, updates slower and require disk space
- Where clause
  - **Exact match - hash index**
  - **range query - tree index**
    - cluster may help
- Multi-attribute
  - order of attribute is important for range queries
  - can sometimes enable **index-only strategies** - clustering is not important

- Example

```sql
SELECT E.dno FROM Emp E WHERE E.age>40
/*
ragne so b+ tree index on
how selective -> if most people age > 40 not that selective so not worth
is the index cluster?
*/
SELECT E.dno, COUNT(*) FROM Emp E WHERE E.age>10 GROUP BY E.dno
/*
consider the Group by
if many age > 10 -> not selective -> better not create index on age
group by dno, so Cluster E.dno index may be better -> dont even have to
fetch, use only index very fast
*/
SELECT E.dno FROM Emp E WHERE E.hobby='Stamps'
/*
equality search but hobby not uniqe
clustering on E.hobby helps!
*/
```

Indexs with Composite Search Keys

- **in goodnote😭**