Relational Algebra

Basic Operation

Select σ

Project Π

Union $\bigcup$

Set different $-$

Product $\times$

Rename ρ

Division $/$

Example

- $Select\ Relation\ a = b\ and\ d > 5 = \sigma_{a=b \bigwedge d>5}(R)$
- $\Pi_{f\_name,\ l\_name}(\sigma_{dno=4 \bigwedge salary>25000}(Employee))$
- $Rename\ E.sid\ to\ C = \rho(C(sid \to identity),\ E)$
- $Condition\ Join = R \bowtie_c S = \sigma_c(R \times S)$

SQL

Basic SQL Query

```sql
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```

- Equal Join with tables

```sql
SELECT S.sname
FROM Sailors S, Reserves R, Boat B
WHERE S.sid = R.sid AND R.bid = B.bid
```

- Regex in SQL

```sql
SELECT  S.age, age1=S.age-5, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname LIKE ‘B_%B’
```

`LIKE` $for\ string\ matching, ''\ \_\ ''\ means\ any\ characters, ''\ \%\ ''\ means\ arbitary$

- **Set**-manipulation construction
    - `UNION` ⋃
      `INTERSECTION` ⋂
      `EXCEPT` —

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='green'
```

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
EXCEPT
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='green'
```

- Correlated Nested Query

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS(SELECT * FROM Reserves R WHERE R.bid=103 AND S.sid=R.sid)
```

- `EXISTS` to test for nonempty
- `IN` operator specified **multiple values** in `WHERE` clause

```
SELECT B.bname
FROM Boats B
WHERE B.color IN ('red', 'blue', 'green')
```

- **Set-comparison** operators
- `ops ANY` or `ops ALL`

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT  S2.rating FROM  Sailors S2 WHERE
              S2.sname='Horatio')
```

- Division in SQL

```sql
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS
      ((SELECT B.bid
        FROM Boats B)
        EXCEPT
       (SELECT R.bid
        FROM Reserves R
        WHERE R.sid=S.sid))
```

SQL Aggregate Operators	Description
`COUNT(A)`	Number of values in A column
`SUM(A)`	Sum of all values in A column
`AVG(A)`	Average of all values on A column
`MAX(A)`	Maximum value in the A column
`MIN(A)`	Minimum value in the A column

- Correct way to use the above operators

```sql
SELECT S.name
FROM Sailors S
WHERE S.age > (SELECT MAX (S2.age)
               FROM Sailors S2
               WHERE S2.rating = 10)
```

- `GROUP BY` similar as for-loop

```sql
SELECT S.rating, MIN(S.age)
FROM Sailors S
GROUP BY S.rating

For i = 1, 2, ..., 10:
    SELECT MIN(S.age)
    FROM Sailors S
    WHERE S.rating = i
```

- Columns appeared in `GROUP BY` should use `HAVING`
- `CREATE VIEW` for creating virtula table based on result SQL statement

```sql
CREATE VIEW Temp AS
  SELECT S.rating, AVG (S.age) AS avgage
  FROM Sailors S
  GROUP BY S.rating

DROP VIEW temp
```

Schema Refinement

Functional Dependencies

```
Let X and Y be nonempty sets of attributes in R
An instance r of R satisfies the FD X->Y if
If t1.X = t2.X, then t1.Y = t2.Y
```

Trivially Preserved

```
If any two row never have the same value for a in a->b
Then a->b is trivially preserved
```

Trivially Dependency

```
If right hand side of arrow is subset of that on left hand side a->b
Then a->b is a trivial dependency
```

Closure of set

- Given a set F, the set of all FDs implied is called the closure of F, denoted as F^+

Armstrong's Axioms and additional rules

- **Reflexivity**: $if\ Y \subseteq X,\ then\ X \to Y$

  Augmentation: $if\ X \to Y,\ then\ XZ \to YZ$

  Trasitivity: $if\ X \to Y\ and\ Y \to Z,\ then\ X \to Z$

  Union: $if\ X \to Y\ and\ X \to Z,\ then\ X \to YZ$

  Decomposition: $if\ X \to YZ,\ then\ X \to Y\ and\ X \to Z$

Boyce-Codd Normal Form (BCNF)

$R\ -\ a\ relation\ schema$

$F\ -\ set\ of\ functional\ dependencies\ on\ R$

$R\ is\ in\ BCNF\ if\ for\ any\ X \to A\ in\ F,$

- $X \to A\ is\ a\ trivial\ functional\ dependency,\ i.e.\ A \subseteq X$

OR

- $X \textit{ is a superkey for } R$

Third Normal Form

- If R is in BCNF, then it is also in 3NF since $3NF \subset BCNF$

$R \ - \ a\ relation\ schema$

$F \ - \ set\ of\ functional\ dependencies\ on\ R$

$R\ is\ in\ 3NF\ if\ for\ any\ X \rightarrow A\ in\ F,$

- $X \rightarrow A\ is\ a\ trivial\ functional\ dependency,\ i.e.\ A \subseteq X$

  OR

- $X\ is\ a\ superkey\ for\ R$

  OR

- $A\ is\ part\ of\ some\ key\ for\ R$

Decomposition

1. Remove the rule X → A that violates the condition from relation schema R
2. Create a new realtional schema XA

Lossless Join Decomposition

$R \ - \ a\ relation\ schema$

$F \ - \ set\ of\ functional\ dependencies\ on\ R$

$The\ decomposition\ of\ R\ into\ relations\ with\ attribute\ sets\ R_1,\ R_2\ is\ lossless - join\ iff$

$(R_1 \bigcap R_2) \rightarrow R_1 \in F^+$

OR

$(R_1 \bigcap R_2) \rightarrow R_2 \in F^+$

$R_1 \bigcap R_2\ is\ a\ superkey\ for\ R_1\ or\ R_2$

Dependency preserved

$(F_1 \bigcup F_2)^+ = F^+$

- Possible to obtain lossless-join decomposition into collection of BCNF relation schemas

  $non - BCNF \rightarrow BCNF$ but **NOT** guaranteed dependency-preserving

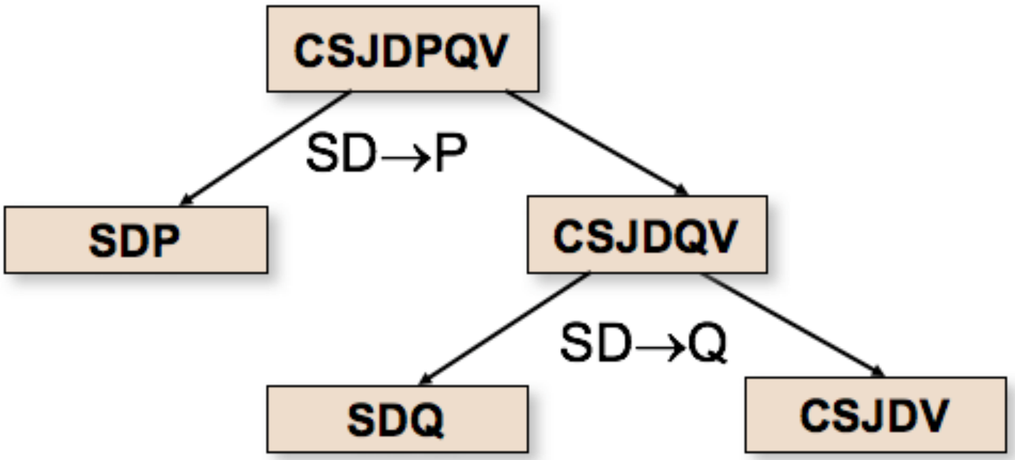
- Always exists a dependency-preserving, lossless-join decomposition into collection of 3NF relation schemas

  $non - 3NF \rightarrow 3NF$

BCNF Decomposition

Suppose $X \to A$ is a FD that violates the BCNF condition

1. Decompose R into XA and $R - A$
2. Repeat until all relations become BCNF

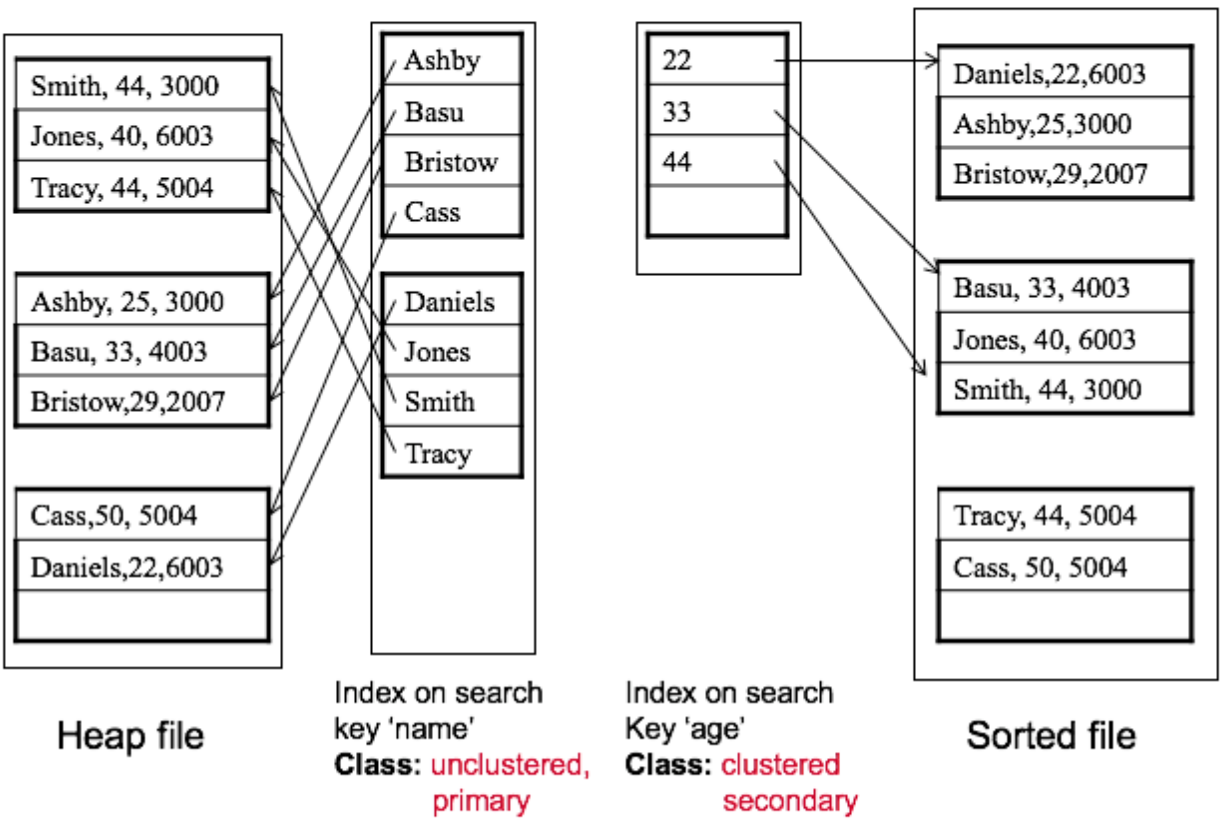


Canonical Cover

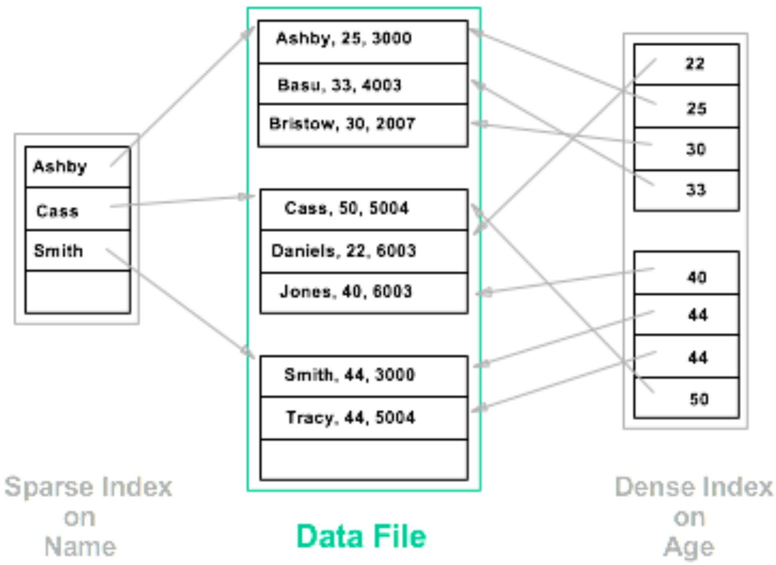
- A **minimal and equivalent** set of functional dependency

Storage and Index

- Index on file speeds up selections on **search key fields**
- Search key can be any subset of fields of relation



Primary index	**Contains primary key in search key**
Secondary index	**Does not contain primary key in search key**
Clustered index	**Order of data records close to order of data entries**
Unclustered index	
Dense index	**At least one data entry per search key value**
Sparse index	**Every sparse index is clustered**



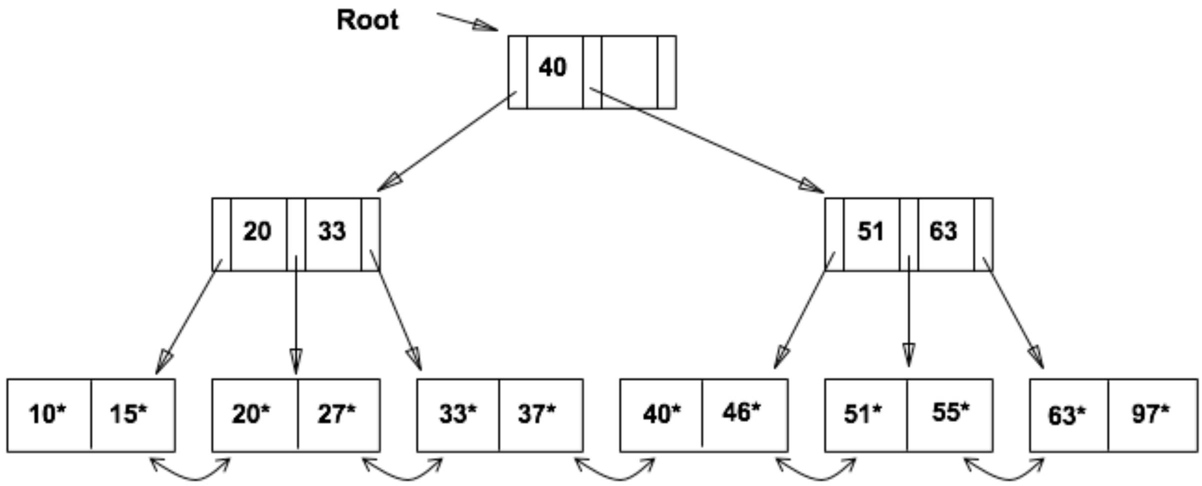
- **Unclustered** must be also in **dense**
  - Primary: each data entry k^* points to **single record** that contains k
  - Secondary: each data entry k^* points to **all records** that contains k
- **Clustered** must be also in **sparse**
  - Sort both data file and index file on search key
  - Each data entry k^* points to the **first record** that contains k
  - Overflow pages may be needed for inserting, so the order is **closed to** sorted

Composite Search Keys

Equality query	**Every field value is equal to a constant value**
Range query	**Some field value is not a constant**

Tree-Structured Indexing

- Example of B+ tree with **order 1**



$For\ root\ node,\ we\ require\ 1 \leq n \leq 2d,\ where\ d\ is\ the\ order$

$For\ non-root\ node,\ we\ require\ d \leq n \leq 2d,\ where\ d\ is\ the\ order$

Cost for searching in B+ tree

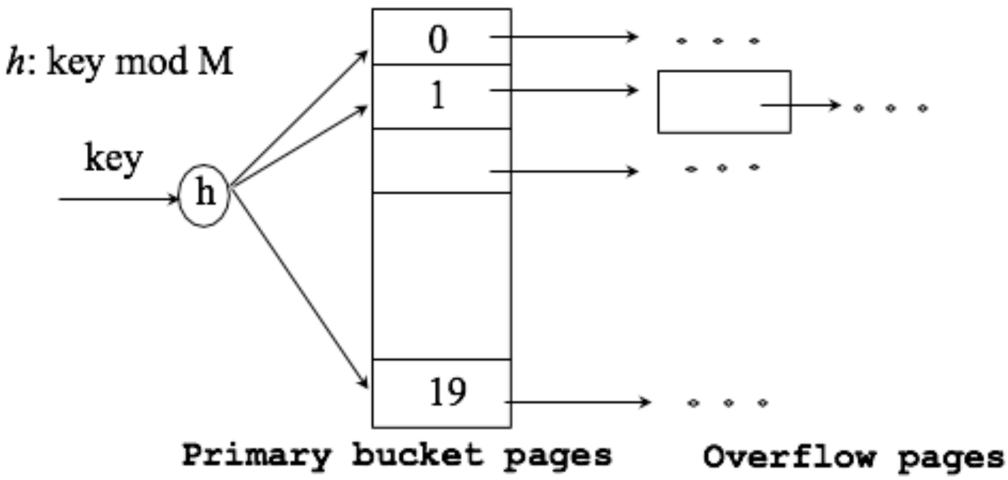
- Let h be the height of B+ tree, then we have to access $h+1$ pages to reach leaf node
- Let f be the average number of pointers in node (*fanout* for internal node)
  - Level 1 with height 0 = 1 page = f^0 page
  - Level 3 with height 2 = $f \times f$ page = f^2 page
- Suppose there are d data entries, so there are $\frac{d}{(f-1)}$ leaf nodes and $h = log_f(\frac{d}{f-1})$
- Example for calculation
  - Typical order = 100, Typical fill-factor = 67%
  - Average fanout f = $\frac{100}{67\%} = 133$
  - Given there are 10000000 data entries, $h = log_{133}(\frac{10000000}{133-1}) < 4$
  - Therefore, the cost is 5 pages read

Extensible Hashing

- Given a search key value k, we can find the bucket where data entry $k*$ is stored
- The value of has function $h(k)$ is address of desired bucket

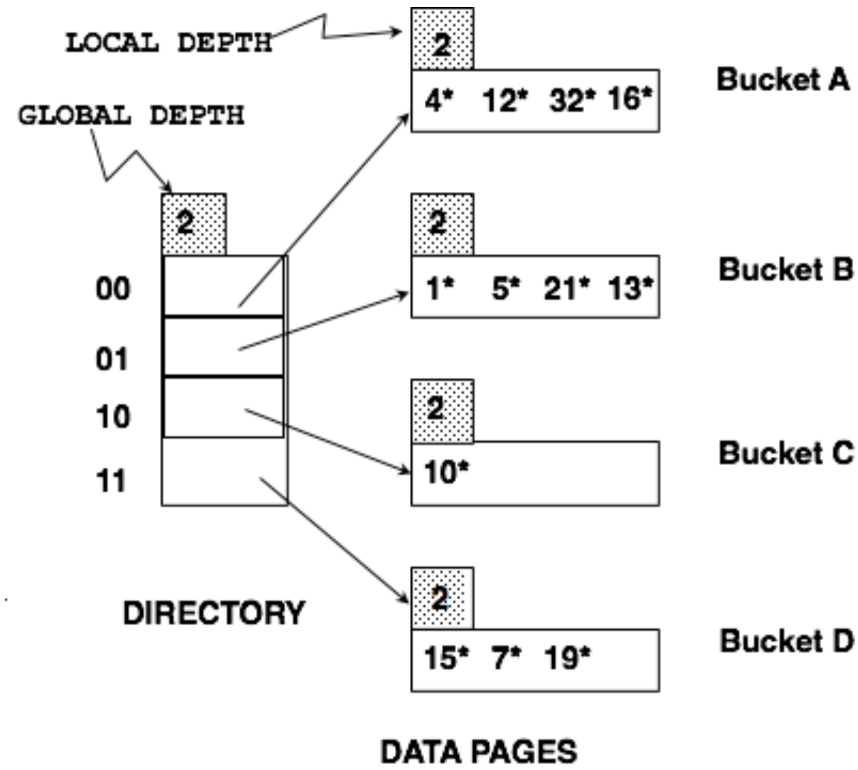
Hash-based indexes are the best for **equality selections** and they do **NOT** support range searches

Static Hashing



- **Long overflow chains** can be developed and degraded performance

Extendsible Hashing

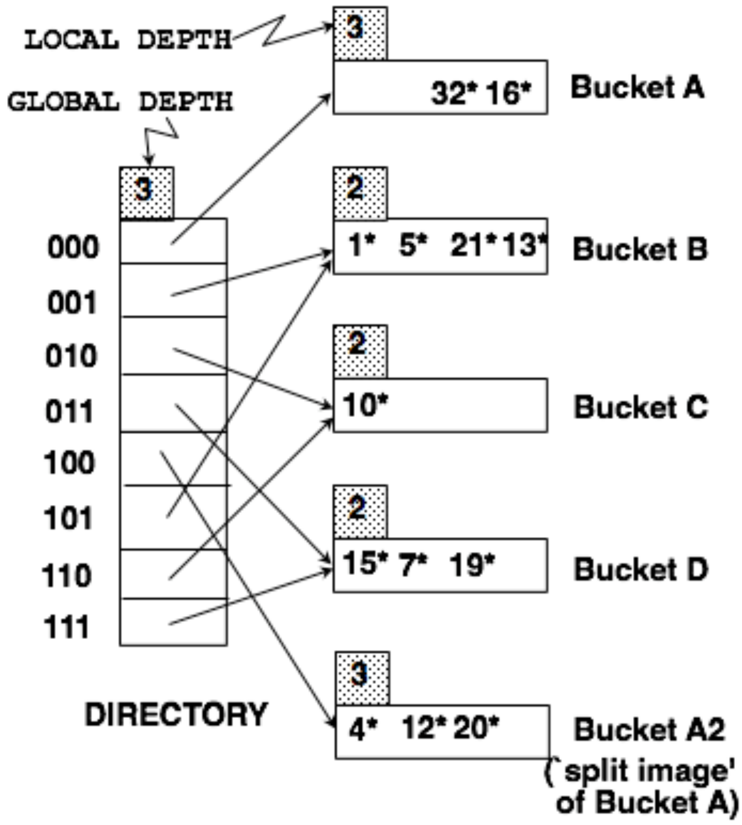


- To avoid re-hashing to re-organize file by doubling numbers of buckets, use directory of points to buckets
- Doubling here means increasing size of directories

$Suppose\ we\ have\ a\ hash\ function\ h(r)\ and\ directory\ is\ array\ of\ size\ 4$

$To\ find\ the\ bucket\ for\ r,\ take\ last\ x\ bits\ of\ h(r)\ where\ x\ is\ number\ of\ global\ depth$

- For inserting 20 in the above example, it will cause overflow and directory doubling is required



- Split Bucket A into 2 buckets and we compare 3rd bit from right in $h(r)$ to decide A or A_2
- Other buckets will remain unchange and 2 directories are pointed to bucket for $Global\ Depth > Local\ Depth$
- **Least siginificant bits** are used in directory to allow for doubling via copying

Query Evaluation