Hash Indexes: Chap. 11

HW1 5 #10

- 10. For every supplier that only supplies green parts, print the name of the supplier and the total number of parts that she supplies.
- Here, clearly need to group by sid
- First look at: suppliers that supply only green parts
  - All suppliers except suppliers that supply non-green parts
  - Mysql doesn't implement EXCEPT: not in Entry SQL-92
  - So use: all suppliers NOT IN (suppliers that supply non-green parts)
  - Select sid from suppliers where sid not in (select sid from catalog c, parts p where c.pid = p.pid and p.color <> 'Green')

select s.sname, count (pid) from suppliers s, catalog c, parts p where <join conditions> and s.sid in (above query) group by s.sid, s.sname; --add sname since in select list

HW1 #11

11. For every supplier that supplies a green part and a red part, print the name and price of the most expensive part that she supplies.
- Need to avoid INTERSECT as well as EXCEPT for mysql
  - Suppliers of green parts where sid IN (suppliers of red parts)
- Price of most expensive part for a certain supplier, say sidx:
  - Select max(c.cost) from catalog where sid = sidx
- Name of most expensive part for each sid:
  - Select c.sid, p.pname from catalog c, parts p where c.pid = p.pid and c.cost = (select max(c.cost) from catalog where sid = c.sid);
- Putting it together, using sname (could just use sid here)

select distinct s.sname, p.pname, c.cost from parts p, suppliers s, catalog c
  where p.pid=c.pid and s.sid=c.sid and c.cost=
    (select max(c1.cost) from catalog c1 where c1.sid=c.sid) and c.sid in
      (Suppliers of green parts where sid IN (suppliers of red parts))

HW1 #11, continued

Final query: Using sname (could just use sid here)

select distinct s.sname, p.pname, c.cost from parts p, suppliers s, catalog c
  where p.pid=c.pid and s.sid=c.sid and c.cost=
    (select max(c1.cost) from catalog c1 where c1.sid=c.sid) and c.sid in
      (select c2.sid from catalog c2, parts p2
        where c2.pid=p2.pid and p2.color='Red' and c2.sid in
          (select c3.sid from catalog c3, parts p3
            where c3.pid=p3.pid and p3.color='Green'));

Without the "distinct", we get two identical rows of output because of a part that comes in two colors at the same price and with the same part name.

We could use "group by pname" instead or "group by pname, sname"

Note that "group by sid, pid" is not needed, because there is only one row in the big join for each (sid, pid) pair.

HW 2 Bench Table

- Table of 1M rows, Columns of different "cardinalities"

```
CREATE TABLE BENCH (
        KSEQ integer primary key,
        K500K integer not null, K250K integer not null,
        K100K integer not null, K40K integer not null,
        K10K integer not null,  K1K integer not null,
        K100 integer not null,   K25 integer not null,
        K10 integer not null,  K5 integer not null,
        K4 integer not null,    K2 integer not null,
        S1 char(8) not null,     S2 char(20) not null,
        S3 char(20) not null,  S4 char(20) not null,
        S5 char(20) not null,  S6 char(20) not null,
        S7 char(20) not null,  S8 char(20) not null)
tablespace cs634test storage(initial 1 M next 1 M );
```

- **Column K500K has 500K different values 1, 2, ..., 500,000**
- **Column K2 has 2 different values 1,2 (cardinality 2)**

Table Bench is in tablespace cs634test

```
create tablespace cs634test
datafile '/disk/sd1e/data/oracle-10.1/dbs2/cs63401.dbf'
size 1 G
default storage ( initial 1 M next 1 M);
```

- Shows how a disk file on disk sd1 becomes part of the database. Oracle makes the file based on this spec.

- Unfortunately, MySQL v. 5.6 does not allow this simple way of adding a file (v. 5.7 does)

Loading table bench

▸ First a C program creates a datafile bench.dat:

```
dbs2(10)% head -3 bench.dat
1 16808 225250 50074 23659 8931 273 45 4 4 5 1 2 12345678
12345678900987654321 12345678900987654321
12345678900987654321 12345678900987654321
12345678900987654321 12345678900987654321
12345678900987654321
2 484493 243043 7988 2504 2328 730 41 13 4 5 2 2 12345678
12345678900987654321 12345678900987654321
12345678900987654321 12345678900987654321
12345678900987654321
3 129561 70934 93100 279 1817 336 98 2 3 3 3 2 12345678
12345678900987654321 12345678900987654321
12345678900987654321 12345678900987654321
12345678900987654321
dbs2(11)% pwd        (show this is on local disk sd0)
/disk/sd0d/tools/sun4-sos5/cs634test/setq
```

▸

Then a bulk load

```
topcat$ more bench.ctl
load data
replace
into table bench
fields terminated by " "
(KSEQ, K500K, K250K, K100K, K40K, K10K, K1K, K100, K25,
K10, K5, K4, K2, S1, S2, S3, S4, S5, S6, S7, S8)
```

▸ Note this builds the PK index, not clustered.
▸ The data file is on disk sd0, the tablespace on disk sd1, so this load uses both disks at top speed.
▸ Would be horribly slower if the data file was on networked disk.
▸ The load took about 10 minutes.
▸ That's 210 MB data read in 600 s, or about 350 KB/s read rate.

▸

Then add secondary indexes on some columns

```
CREATE INDEX k500kin ON bench (k500k)
    storage (initial 1 M next 1 M) pctfree 5 tablespace CS634TEST;
COMMIT WORK;
CREATE  INDEX k100kin on bench (k100k)
    storage (initial 1 M next 1 M) pctfree 5 tablespace CS634TEST
COMMIT WORK;
CREATE INDEX k10kin on bench (k10k)
    storage (initial 1 M next 1 M) pctfree 5 tablespace CS634TEST;
COMMIT WORK;
CREATE INDEX k100in on bench (k100)
    storage (initial 1 M next 1 M) pctfree 5 tablespace CS634TEST;
COMMIT WORK;
CREATE INDEX k10in on bench (k10)
    storage (initial 1 M next 1 M) pctfree 5 tablespace CS634TEST;
COMMIT WORK;
CREATE INDEX k4in on bench (k4)
    storage (initial 1 M next 1 M) pctfree 5 tablespace CS634TEST;
COMMIT WORK;
```
We could make a tablespace on sd0 for these indexes and get better performance for
some queries. This took a minute or so for each index.

▸

Final Steps for Bench Table

Analyze the table to get stats for the query processor

```
analyze table bench compute statistics
for table for all indexes;
```

Make it publicly readable:
```
grant select on table bench to public;
```

▸

Try it out from another (non-priv) account

```
dbs2(20)% sqlplus cs636test/…
SQL> select count(*) from eoneil.bench;
  COUNT(*)
----------
   1000000
SQL> select tablespace_name from all_tables
  where table_name = 'BENCH';

TABLESPACE_NAME
-----------------------------
CS634TEST

SQL> select index_name,index_type, uniqueness from all_indexes where
table_name='BENCH';
INDEX_NAME                    INDEX_TYPE                 UNIQUENES
----------------------------- -------------------------- ---------
SYS_C00549655                 NORMAL                     UNIQUE
K500KIN                       NORMAL                     NONUNIQUE
K100KIN                       NORMAL                     NONUNIQUE
…
```

▸

Overview

▸ **Hash-based** indexes are best for **equality selections**
  ▸ *Cannot* support range searches, except by generating all values
  ▸ Static and dynamic hashing techniques exist

▸ Hash indexes not as widespread as B+-Trees
  ▸ Some DBMS do not provide hash indexes
  ▸ But hashing still useful in query optimizers (DB Internals)
  ▸ E.g., in case of equality joins

▸ As for tree indexes, 3 alternatives for data entries $k*$
 ▸ Choice orthogonal to the indexing technique

▸

Hashing in Memory and on Disk

- The hash table may be located in memory, supporting fast lookup to records on disk, or even on disk, supporting fast access to further disk.
- In fact, a disk-resident hash table that is in frequent use ends up being in memory because of the memory "caching" of disk pages in the file system.

| keys →	hash table →	Data records	Example
memory	memory	memory	typical HashMap apps
memory	memory	disk	use HashMap to hold disk record locations as values
memory	disk	disk	hashed files, some database tables

Static Hashing

▸ Number of buckets *N* fixed, each with primary, overflow pages
  ▸ primary pages are allocated sequentially
  ▸ overflow pages may be needed when file grows
  ▸ Buckets contain data entries
▸ **Hash value: h**(*k*) mod N = bucket for data entry with key *k*

Primary bucket pages   **Overflow pages**
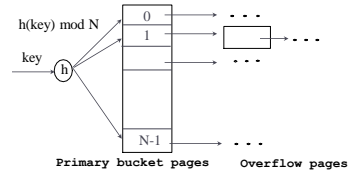
Static Hashing

▸ Hash function is applied on *search key* field
  ▸ Must distribute values over range 0 ... N-1.
  ▸ **h(*key*) = (a * *key* + b)** is a typical choice (for numerical keys)
  ▸ a and b are constants, chosen to "tune" the hashing, and prime
  ▸ Example: h(key) = 37*key + 101
▸ Hash function for string keys? A tricky subject, easy to go wrong
  ▸ See Wikipedia article https://en.wikipedia.org/wiki/Hash_function
  ▸ Algorithm used by Perl: https://en.wikipedia.org/wiki/Jenkins_hash_function

Data entries can be full rows (Alt (1))

Primary bucket pages   **Overflow pages**
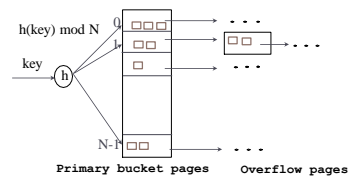
▸ Primary pages are sequential on disk, so full table scan is fast if not too many overflow pages, or overflow pages are also sequential
▸ Is a clustered index

Data entries can be (key, rid(s)) (Alt (2,3))

Requires data sorted in hash-value order
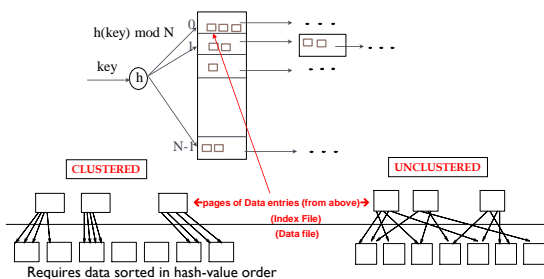
Static Hashing

▸ Works well if we know how many keys there can be
  ▸ Then we can size the primary page sequence properly: keep it under about half full
  ▸ Can have "collisions": two keys with same hash value
▸ But when file grows considerably there are problems
▸ Long overflow chains develop and degrade performance
  ▸ Example: loader took over an hour to load a big program
  ▸ Found it was hashing using 1000-spot hash table for global symbols! One line edit→ solved the problem.
▸ General Solution: Dynamic Hashing, 2 contenders described:
 ▸ Extendible Hashing
  ▸ Linear Hashing

Extendible Hashing

▸ Main Idea: when primary page becomes full, double the number of buckets
  ▸ But reading and writing all pages is expensive
  ▸ Use directory of pointers to buckets
  ▸ Double the directory size, and only split the bucket that just overflowed!
  ▸ Directory much smaller than file, so doubling it is cheap
  ▸ There are no overflow pages (unless the same key appears a lot of times, i.e., very skewed distribution – many duplicates)

▸

Extendible Hashing Example

▸ Directory is array of size 4
▸ Directory entry corresponds to last two bits of hash value

▸ If **h**(k) = 5 = binary 101, it is in bucket pointed to by 01

▸ Insertion into non-full buckets is trivial
▸ Insertion into full buckets requires split and **directory doubling**
▸ E.g., insert h(k)=20

Insert h(k)=20 (Causes Doubling)

Use last 3 bits in split bucket!

▸
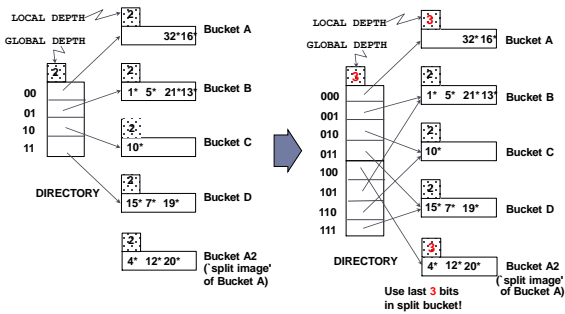
Global vs Local Depth

▸ *Global depth of directory:*
  ▸ Max # of bits needed to tell which bucket an entry belongs to
▸ *Local depth of a bucket:*
  ▸ # of bits used to determine if an entry belongs to this bucket
▸ When does bucket split cause directory doubling?
  ▸ Before insert, *local depth* of bucket = *global depth*
  ▸ Insert causes *local depth* to become > *global depth*
  ▸ Directory is doubled by *copying it over*
    ▸ Use of least significant bits enables efficient doubling via copying of directory
▸ Delete: if bucket becomes empty, merge with `split image'
  ▸ If each directory element points to same bucket as its split image, can halve directory

▸

Directory Doubling

Why use least significant bits in directory?
It allows for doubling via copying!
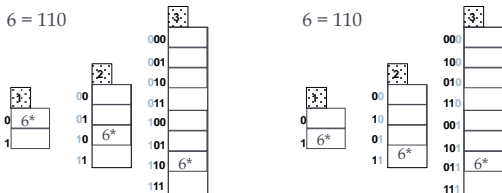
Least Significant   vs.   **Most Significant**

▸

Extendible Hashing Properties

▸ If directory fits in memory, equality search answered with one I/O; otherwise with two I/Os
  ▸ 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records
    ▸ (That's 100MB/(100 bytes/rec) = 1M recs)
  ▸ 25,000 directory elements will fit in memory
    ▸ (That's assuming a bucket is one page, 4KB = 4096 bytes, so can hold 4000 bytes/(100 bytes/rec)= 40 recs, plus 96 bytes of header, so 1Mrecs/(40 recs/bucket) = 25,000 buckets, so 25,000 directory elements)
  ▸ Multiple entries with same hash value cause problems!
    ▸ These are called collisions
    ▸ Cause possibly long overflow chains

▸

Linear Hashing

- Dynamic hashing scheme
- Handles the problem of long overflow chains
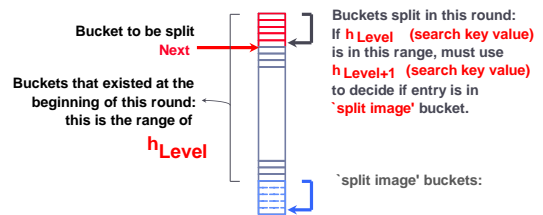 - But does not require a directory!
  - Deals well with collisions!

▸

Linear Hashing

- Main Idea: use a family of hash functions $h_0, h_1, h_2, ...$
  - $h_i(key) = h(key) \bmod(2^i N)$
    - N = initial number of buckets
  - If $N = 2^{d_0}$, for some d_0, h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$
  - h_{i+1} doubles the range of h_i (similar to directory doubling)
  - Example:
  - N=4, conveniently a power of 2
  - $h_i(key) = h(key) \bmod(2^i N) = h(key)$, last 2+i bits of key
  - $h_0(key) =$ last 2 bits of key
  - $h_1(key) =$ last 3 bits of key
  - …

▸

Linear Hashing: Rounds

- During round 0, use h_0 and h_1
- During round 1, use h_1 and h_2
- …
- Start a round when some bucket overflows
  - (or possibly other criteria, but we consider only this)
- Let the overflow entry itself be held in an overflow chain
- During a round, split buckets, in order from the first
- Do one bucket-split per overflow, to spread out overhead
- So some buckets are split, others not yet, during round.
- Need to track division point: Next = bucket to split next

▸

Overview of Linear Hashing

Buckets split in this round:
If h_{Level} (search key value) is in this range, must use $h_{Level+1}$ (search key value) to decide if entry is in `split image' bucket.

Bucket to be split
Next

Buckets that existed at the beginning of this round:
this is the range of
h_{Level}

`split image' buckets:

Note this is a "file", i.e., contiguous in memory or in a real file.

▸

Linear Hashing Properties

- Buckets are split round-robin
  - Splitting proceeds in `rounds'
    - Round ends when all N_R initial buckets are split (for round R)
    - Buckets 0 to Next-1 have been split; Next to N_R yet to be split.
  - Current round number referred to as *Level*
- Search for data entry *r* :
  - If $h_{Level}(r)$ in range `Next to N_R', search bucket $h_{Level}(r)$
  - Otherwise, apply $h_{Level+1}(r)$ to find bucket

▸

Linear Hashing Properties

- Insert:
  - Find bucket by applying h_{Level} or $h_{Level+1}$ (based on *Next* value)
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - Split *Next* bucket and increment *Next*
- Can choose other criterion to trigger split
  - E.g., occupancy threshold
- Split round-robin prevents long overflow chains

▸

Example of Linear Hashing

- On split, $h_{Level+1}$ is used to re-distribute entries.


```
            Level=0, N=4                                        Level=0

      h     h    PRIMARY                          h    h    PRIMARY      OVERFLOW
      1     0    PAGES                            1    0    PAGES        PAGES
          Next=0                                      Next=1
      000   00   32* 44* 36*                      000  00   32*
                              Data entry r
      001   01   9* 25* 5*    with h(r)=5         001  01   9* 25* 5*
                              Primary
      010   10   14* 18* 10* 30*  bucket page     010  10   14* 18* 10* 30*

      011   11   31* 35* 7* 11*                   011  11   31* 35* 7* 11*   →  43*

                (The actual contents              100  00   44* 36*
                of the linear hashed
                file)                                      After inserting 43*
```
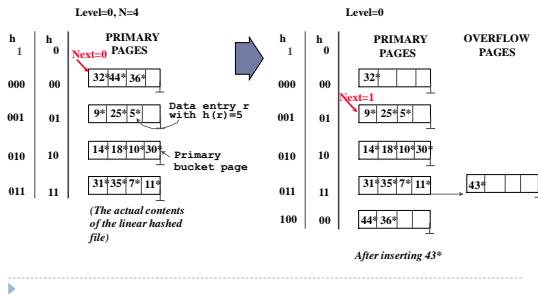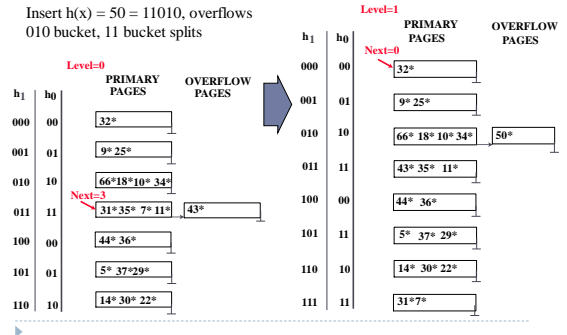
End of a Round

Insert h(x) = 50 = 11010, overflows
010 bucket, 11 bucket splits


```
                          Level=0                                                  Level=1
                                                                    h    h    PRIMARY         OVERFLOW
       h    h    PRIMARY      OVERFLOW                              1    0    PAGES            PAGES
       1    0    PAGES        PAGES                                     Next=0
                                                                   000  00   32*
      000   00   32*
                                                                   001  01   9* 25*
      001   01   9* 25*
                                                                   010  10   66* 18* 10* 34*   50*
      010   10   66* 18* 10* 34*
                 Next=3                                            011  11   43* 35* 11*
      011   11   31* 35* 7* 11*   43*
                                                                   100  00   44* 36*
      100   00   44* 36*
                                                                   101  11   5* 37* 29*
      101   01   5* 37* 29*
                                                                   110  10   14* 30* 22*
      110   10   14* 30* 22*
                                                                   111  11   31* 7*
```

Advantages of Linear Hashing

- Linear Hashing avoids directory by:
  - splitting buckets round-robin
  - using overflow pages
  - in a way, it is the same as having directory doubling gradually

- Primary bucket pages are created in order
- Easy in a disk file, though may not be really contiguous
- But hard to allocate huge areas of memory

Summary

- Hash-based indexes: best for equality searches, (almost) cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. *(Duplicates may require overflow pages.)*
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas in the early part of a round.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed
- Need a good hash function!

Indexes in Standards

- SQL92/99/03 does not standardize use of indexes
- (BNF for SQL2003)
  - But all DBMS providers support it
  - X/OPEN actually standardized CREATE INDEX clause
  CREATE [UNIQUE] INDEX indexname ON tablename
      (colname [ASC | DESC] [,colname [ASC | DESC] ,. . .]);
- ASC|DESC are just there for compatibility, have no effect in any DB I know of.
- Index has as key the concatenation of column names
  - In the order specified

Indexes in Oracle

▸ Oracle supports mainly B+-Tree Indexes
  ▸ These are the default, so just use create index…
  ▸ No way to ask for clustered directly
  ▸ Clustering on PK is available via index-organized tables (IOTs)
    ▸ In this case, the RID is different, affecting secondary index performance
  ▸ Also "table cluster" for co-locating data of tables often joined

▸ Hashing: via "hash cluster"
  ▸ Also a form of hash partitioning supported
  ▸ Also supports bitmap indexes
  ▸ Hash cluster example→

▸

Example Oracle Hash Cluster

CREATE CLUSTER trial_cluster (trialno DECIMAL(5,0))
  SIZE 1000 HASH IS trialno HASHKEYS 100000;
CREATE TABLE trial (trialno DECIMAL(5,0) PRIMARY KEY, ...)
  CLUSTER trial_cluster (trialno);

▸ SIZE should estimate the max storage in bytes of the rows needed for one hash key
▸ Here HASHKEYS <value> specifies a limit on the number of unique keys in use, for hash table sizing. Oracle rounds up to a prime, here 100003. This is static hashing.

▸

Oracle Hash Index, continued

▸ For static hashing in general: rule of thumb—
  ▸ Estimate the max possible number of keys and double it. This way, about half the hash cells are in use at most.
▸ The hash cluster is a good choice if queries usually specify an exact trialno value.
▸ Oracle will also create a B-tree index on trialno because it is the PK. But it will use the hash index for equality searches.

▸

MySQL Indexes, for InnoDB Engine

▸ CREATE [UNIQUE] INDEX *index_name* [*index_type*] ON *tbl_name* (*index_col_name*,...)
▸ *index_col_name*: *col_name* [(*length*)] [ASC | DESC]
▸ *index_type*: USING {BTREE | HASH}

▸ Syntax allows for hash index, but not supported by InnoDB.
▸ For InnoDB, index on primary key is clustered.

▸

Clustered index on PK: choose your PK wisely

▸ Available in Oracle and MySQL, as only kind of clustered B-tree index.
▸ Common PKs are ids, arbitrary, not commonly used in range queries, so not getting the good from the clustered B-tree.
▸ However, a PK is what we say it is for a table, and doesn't need to be minimalistic, just a unique identifier.
▸ So (zipcode, custid) works as a PK and clusters the data by zipcode. Custid is a "uniquifier" here.
▸ Then useful range queries on zipcode run fast.

▸

Indexes in Practice

▸ Typically, data is inserted first, then index is created
▸ Exception: alternative (1) indexes (of course!)
  ▸ Then best to sort first, then load
  ▸ How to sort? Use database: load, sort, dump, load for real
▸ Index bulk-loading is a good idea – recall it is much faster
▸ Delete an index
  DROP INDEX indexname;

▸ Guidelines:
  ▸ Create index if you frequently retrieve less than 15% of the table
  ▸ To improve join performance, index columns used for joins
  ▸ Small tables do not require indexes, except ones for PKs.

▸

Compare B-Tree and Hash Indexes

▸ Dynamic Hash tables have variable insert times
▸ Worst-case access time & best average access time
▸ But only useful for equality key lookups
▸ Note there are bitmap indexes too

▸