Index Tuning

Outline

- Index Utilization
  - Heap Files
  - Definition: Clustered/Non clustered, Dense/Sparse
  - Access method: Types of queries
  - Constraints and Indexes
  - Locking
- Index Data Structures
  - B+-Tree / Hash
  - LSM Tree / Fractal tree
  - Implementation in DBMS
- Index Tuning
  - Index data structure
  - Search key
  - Size of key
  - Clustered/Non-clustered/No index
  - Covering

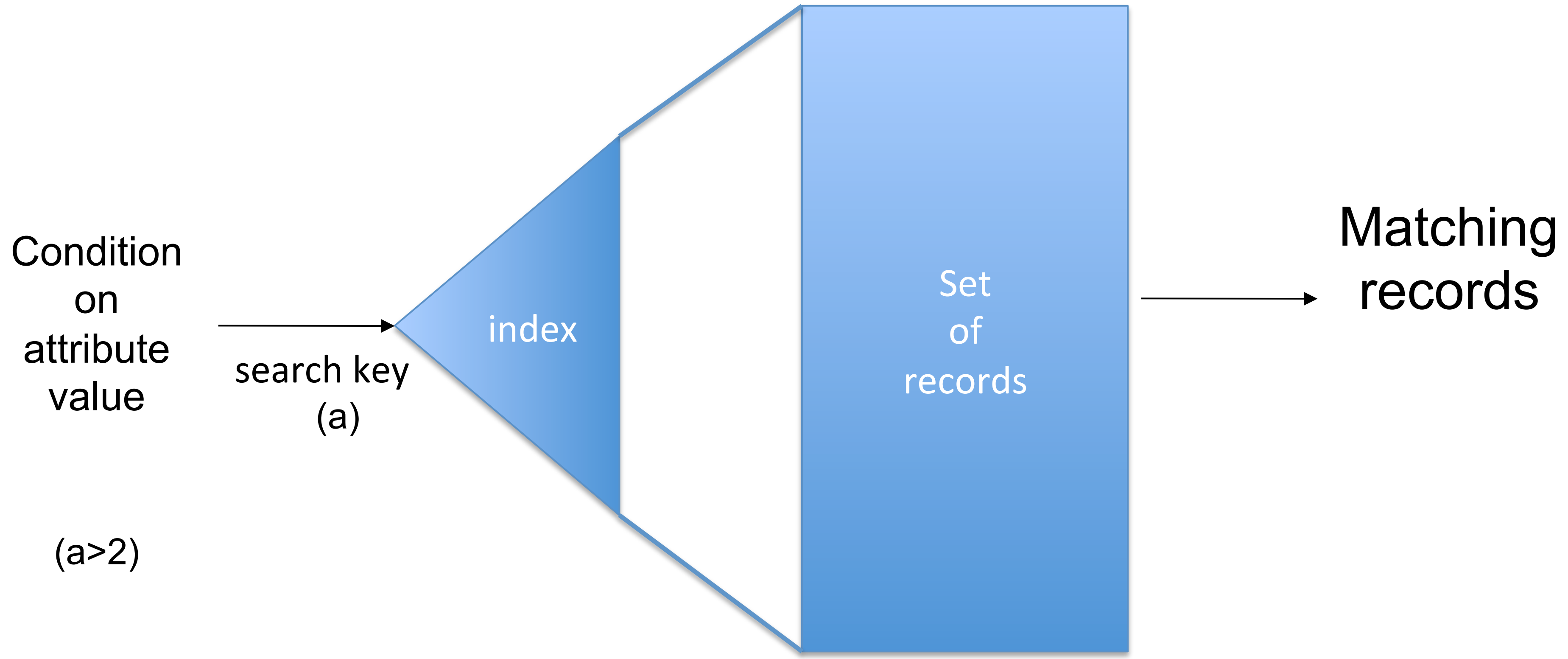
Heap Files

- Rows appended to end of file as they are inserted
  - Hence the file is unordered
  - Good for scan
- Deleted rows create gaps in file
  - File must be periodically compacted to recover space

Index

An index is a data structure that provides efficient access to a set of records

The leaves of the index are (pointers to) records. The internal nodes of the index define its data structure.



Simple SQL Example

```
select ssnum,name
from employees
where hundreds2 > 10;
```

```
create index nc on
employees (hundreds2);
```

```
create index nc1 on
employees (hundreds2, ssnum, name)
```

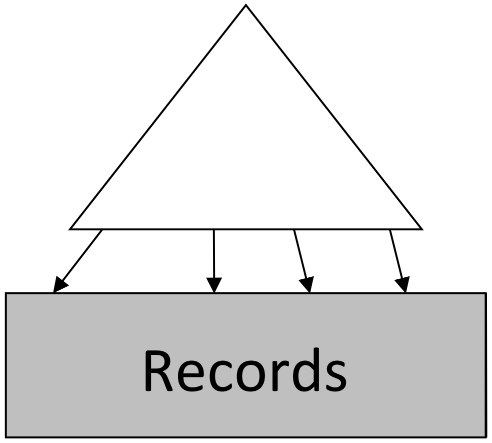
```
create table employees(
        ssnum integer not null,
        name varchar(20) not null,
        lat real, long real,
        hundreds1 real,
        hundreds2 real
);
```

Search Keys

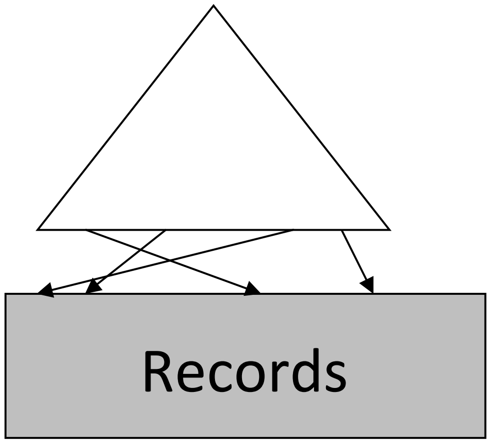
- A (search) key is a <u>sequence</u> of attributes.
  create index i1 on accounts(branchnum, balance);
- Types of keys
  - Sequential: the value of the key is monotonic with the insertion order (e.g., counter or timestamp)
  - Non sequential: the value of the key is unrelated to the insertion order (e.g., social security number)

Clustered / Non clustered index

- Clustered index (primary index)
  - A clustered index on attribute X co-locates records whose X values are *near* to one another.



- Non-clustered index (secondary index)
  - A non clustered index does not constrain table organization.
  - There might be several non-clustered indexes per table.



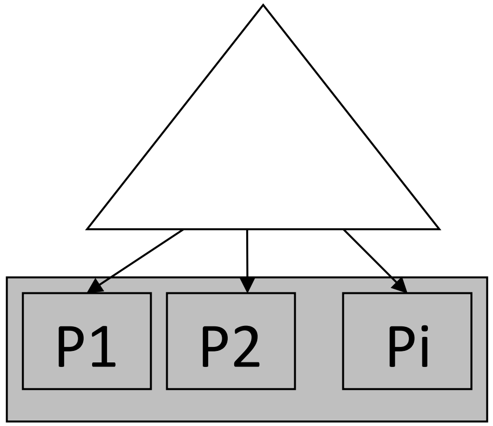
Index-Organized Tables

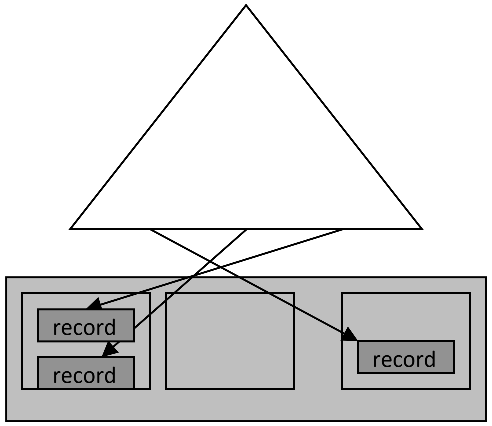
- Clustered indexes organized how data are stored on disk
  - Clustered index and storage can be orthogonal
    - A clustered index can be dropped in which case the table is organized as a heap file
    - A clustered index can be defined on a table (previously organized as a heap table), which is then reorganized
  - Index-organized table
    - The clustered index defines the table organization. It is required when the table is defined. It cannot be dropped.

Dense / Sparse Index

- Sparse index
  - Pointers are associated to pages

- Dense index
  - Pointers are associated to records
  - Non clustered indexes are dense





OLTP: Types of Queries

- **Point Query**

```
SELECT balance
FROM accounts
WHERE number = 1023;
```

- **Multipoint Query**

```
SELECT balance
FROM accounts
WHERE branchnum = 100;
```

- **Range Query**

```
SELECT number
FROM accounts
WHERE balance > 10000;
```

- **Prefix Match Query**

```
SELECT *
FROM employees
WHERE  name = 'Jensen'
    and firstname = 'Carl'
    and age < 30;
```

Tables are in BNF, derived from entities and relationships

OLTP: Types of Queries

- **Extremal Query**

```
SELECT *
FROM accounts
WHERE balance =
  max(select balance from accounts)
```

- **Ordering Query**

```
SELECT *
FROM accounts
ORDER BY balance;
```

- **Grouping Query**

```
SELECT branchnum, avg(balance)
FROM accounts
GROUP BY branchnum;
```

- **Join Query**

```
SELECT distinct branch.adresse
FROM accounts, branch
WHERE
  accounts.branchnum =
  branch.number
and accounts.balance > 10000;
```

OLAP

Types of Data

- Multidimensional data
  - Cube
- Spatio-temporal data
  - Time series
  - Spatial data
  - Sensor data

Types of Queries

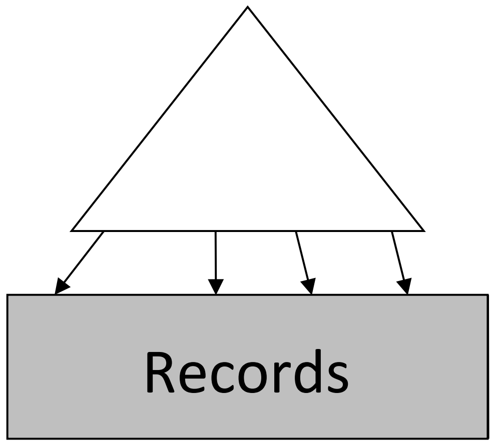
- Cube operators
  - Rollup / Drill down
- Clustering & similarity
  - Nearest neighbors
- Preferences
  - Top K
  - Skyline

See lecture on OLAP tuning

Constraints and Indexes

- Primary Key, Unique
  - A non-clustered index is constructed on the attribute(s) that compose the primary key with the constraint that values are unique.
- Foreign Key
  - By default, no index is created to enforce a foreign key constraint.

Locking



We will review locking in the context of the B+-tree

1. Tree locking
   - Updating a table, requires updating the index (leaves and possibly internal nodes)
   - Concurrent modifications must be scheduled
   - Should locking be used to make conflicts between index writes explicit?
2. Next key locking
   - How can indexes be used to implement a form of predicate locking

Key Compression

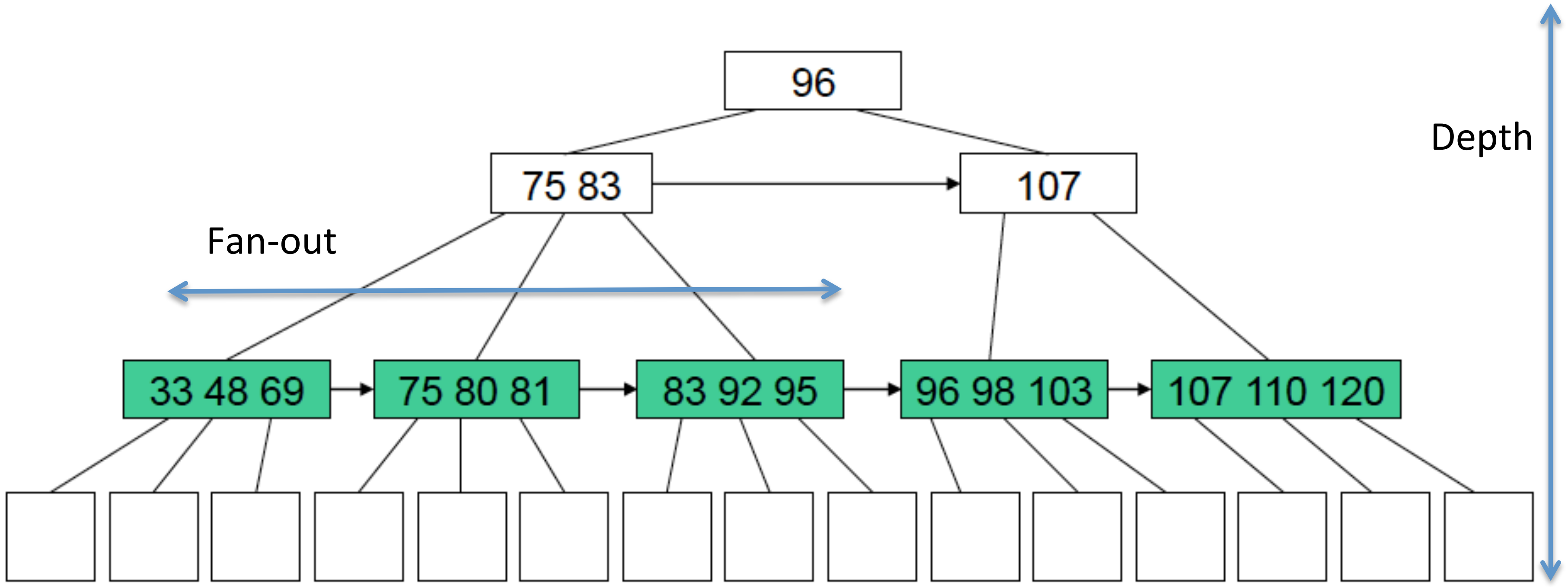
- Prefix compression (Oracle 8+, MySQL): only store that part of the key that is needed to distinguish it from its neighbors: Smi, Smo, Smy for Smith, Smoot, Smythe.
- Front compression (Oracle 5): adjacent keys have their front portion factored out: Smi, (2)o, (2)y. There are problems with this approach:
  - Processor overhead for maintenance
  - Locking Smoot requires locking Smith too.

Data Structures

- Most index data structures can be viewed as trees.
- In general, the root of this tree will always be in main memory, while the leaves will be located on disk.
    - The performance of a data structure depends on the number of nodes in the average path from the root to the leaf.
    - Data structure with high fan-out (maximum number of children of an internal node) are thus preferred.

B+-Tree

A B+-Tree is a balanced tree whose nodes contain a sequence of key-pointer pairs.



B+-Tree

- Nodes contains a bounded number of key-pointer pairs determined by b (*branching factor*)
  - Internal nodes: *ceiling*(b/ 2) <= size(node) <= b
  - Root node:
    - Root is the only node in the tree: 1 <= size(node) <= b
    - Internal nodes exist: 2 <= size(node) <= b
  - Leaves (no pointers):
    - *floor*(b/2) <= number(keys) <= b-1
- Insertion, deletion algorithms keep the tree balanced, and maintain these constraints on the size of each node
  - Nodes might then be split or merged, possibly the depth of the tree is increased.

size(node) = number of key-pointer pairs

B+-Tree Performance #1

- Memory / Disk
  - Root is always in memory
  - What is the portion of the index actually in memory?
    - Impacts the number of IO
    - Worst-case: an I/O per level in the B+-tree!
- Fan-out and tree level
  - Both are interdepedent
  - They depend on the branching factor
    - In practise, index nodes are mapped onto index pages of fixed size
    - Branching factor then depends **on key size** (pointers of fixed size) and page utilization

B+-Tree Performance #2

- Tree maintenance
  - On-line: textbook insertion and deletion algorithms that maintain balanced B+-tree as records are inserted
  - Off-line: inserted/deleted records are inserted in a specific data structure and indexes are modified offline (when the DBA requests it, regularly or when some condition is met).
    - Log-Structured Merge (LSM)-tree: Records inserted in RAM (C0-tree not necessarily organized as a B+-tree – e.g., sorted table), then merged with disk resident C1-tree pages opportunistically.

      See LSM-Tree paper by Par O'neil et al.
    - Heap file: insert buffer in MySQL, default mechanism in DB2.
- Supports well
  - Point queries, Multipoint queries, Range queries, order queries, Extremal queries.

See: VLDB journal survey, ARIES KVL

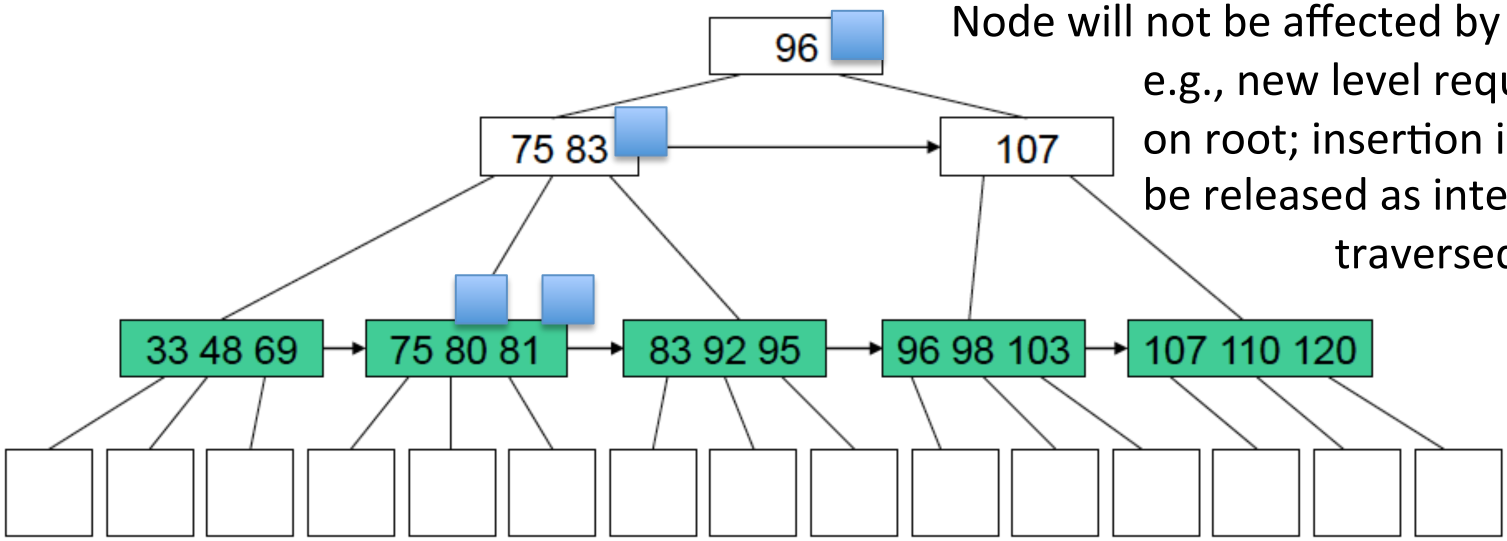
B+-Tree Locking

1. How to avoid locking the whole table and schedule concurrent modifications of the tree?

   Mutexes are used. Top down traversal.

   Mutex released when it is clear that internal Node will not be affected by insertion/deletion

   e.g., new level requires holding mutex on root; insertion in leaf mutexes can be released as internal nodes are traversed.

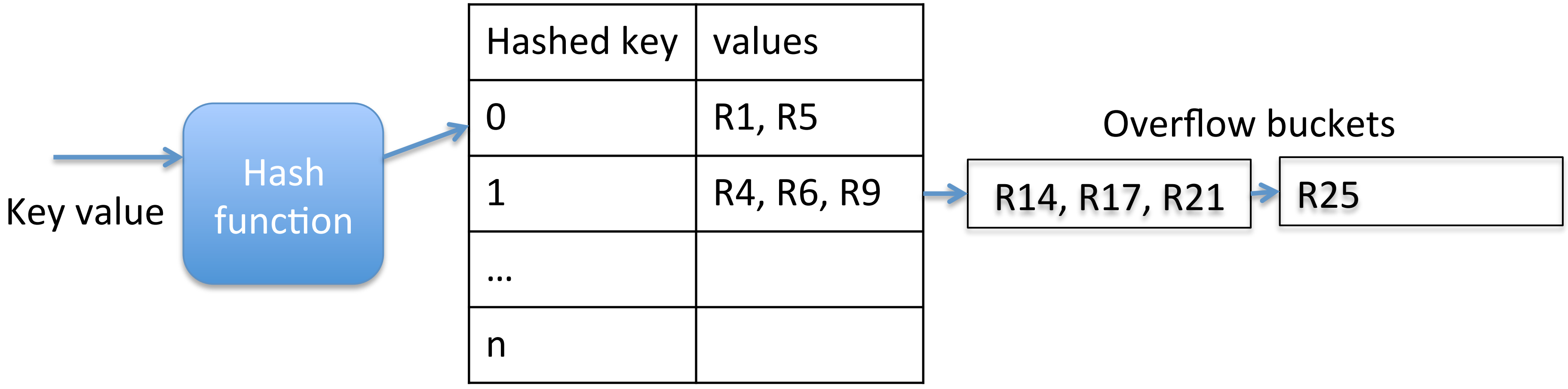


1. How to support predicate-locking?

Next-key locking. A mutex held on a pointer prevents access to all the records that are inserted, or could be inserted, below this pointer or in between this pointer and the next existing pointer. E.g. blue rectangles represent a lock on all records where key >= 80.

Hash Index

- A hash index stores key-value pairs based on a pseudo-randomizing function called a hash function



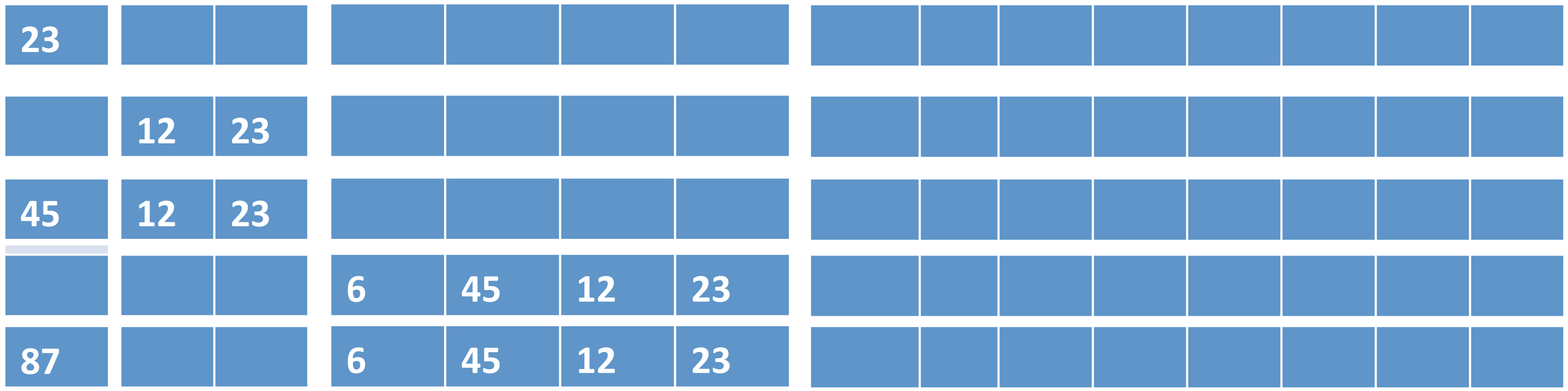
Hashed key	values
0	R1, R5
1	R4, R6, R9
…	
n	

Hash Index Performance

- Memory / Disk
  - Worst case: 1 IO per bucket
  - NOT balanced as number of IOs to reach a record depends on the hash function and key distribution.
- Supports very well
  - Point queries

Fractal Tree Index

- Trees of exponentially increasing size
  - (represented as arrays for ease of representation)
  - Trees are completely full or completely empty
  - Insert into smallest array
  - Merge arrays continuously



Fractal Tree Index Performance

- Few, large, sequential IOs on several trees
  - Scales with large number of insertions (variation of LSM-tree)
  - Potential to leverage SSD parallelism

DBMS Implementation

- Oracle 11g
  - B-tree vs. Hash vs. Bitmap
  - Index-organized table vs. heap
    - Non-clustered index can be defined on both
  - Reverse key indexes
  - Key compression
  - Invisible index (not visible to optimizer – allows for experimentation)
  - Function-based indexes
    - CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);

See: Oracle 11g indexes

DBMS Implementation

- DB2 10
  - B+-tree (hash index only for db2 for z/OS)
  - Non-cluster vs. cluster
  - Key compression
  - Indexes on expression
- SQL Server 2012
  - B+-tree (spatial indexes built on top of B+-trees)
  - Columnstore index for OLAP queries
  - Non-cluster vs. cluster
  - Non key columns included in index (for coverage)
  - Indexes on simple expressions (filtered indexes)

See: DB2 types of indexes, SQLServer 2012 indexes

Clustered Index

Benefits of a clustered index:

1. A sparse clustered index stores fewer pointers than a dense index.
   - This might save up to one level in the B-tree index.
2. A clustered index is good for multipoint queries
   - White pages in a paper telephone book
3. A clustered index based on a B-Tree supports range, prefix, extremal and ordering queries well.

Clustered Index

4. A clustered index (on attribute X) can reduce lock contention:

   Retrieval of records or update operations using an equality, a prefix match or a range condition based on X will access and lock only a few consecutive pages of data

<u>Cost of a clustered index</u>

1. Cost of overflow pages
   - Due to insertions
   - Due to updates (e.g., a NULL value by a long string)

Clustered Index

- Because there is only one clustered index per table, it might be a good idea to replicate a table in order to use a clustered index on two different attributes
  - Yellow and white pages in a paper telephone book
  - Low insertion/update rate

Non-Clustered Index

Benefits of non-clustered indexes

1. A dense index can eliminate the need to access the underlying table through covering.
   - It might be worth creating several indexes to increase the likelihood that the optimizer can find a covering index
2. A non-clustered index is good if each query retrieves significantly fewer records than there are pages in the table.
   - Where is the tipping point wrt heap file? You must experiment on your own system.

Covering Index - defined

- Select name from employee where department = “marketing”
- A priori:
  - Good covering index would be on (department, name)
  - Index on (name, department) less useful.
  - Index on department alone moderately useful.
- Actual impact depends on underlying DBMS.

Index on Small Tables

- Tuning manuals suggest to avoid indexes on small tables
  - If all data from a relation fits in one page then an index page adds an I/O
  - If each record fits in a page then an index helps performance
- However, indexes on small tables allow the DBMS to leverage next key locking and thus reduce contention on small tables.

Key Compression

- Use key compression
  - If you are using a B-tree
  - Compressing the key will reduce the number of levels in the tree
  - The system is not CPU-bound
  - Updates are relatively rare

Index Tuning Wizard

- SQL Server 7 and above, Oracle 9 and above, DB2 V9 and above (design advisor)
- In:
  - A database (schema + data + existing indexes)
  - Trace representative of the workload
- Out:
  - Evaluation of existing indexes
  - Recommendations on index creation and deletion

- The index wizard
  - Enumerates possible indexes on one attribute, then several attributes
  - Traverses this search space using the query optimizer to associate a cost to each index

Summary

1. Use a hash index for point queries only. Use a B-tree if multipoint queries or range queries are used
2. Use clustering
   - if your queries need all or most of the fields of each records returned
   - if multipoint or range queries are asked
3. Use a dense index to cover critical queries
4. Don't use an index if the time lost when inserting and updating overwhelms the time saved when querying