

Database Management



Indexes

Queries at runtime

Suppose we wish to execute a query such as

```
SELECT * FROM Employees WHERE Salary = 38000
```

How do you think it executes your query?

EMPLOYEE									
Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

Queries at runtime

The database loads up the table and scans through each row. (Linear Search)
There it compares the Salary field to see if the value matches what was given.
It collects all the matching results and returns them to you.

So the time it takes to complete the query is linearly dependent on the number of rows - $O(n)$.

EMPLOYEE									
Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1



Let's play a game

Many of you likely are familiar with the “guess a number” example where you are told if you are too high or too low.

For 1 to 100, you simply guess 50.

If too high, your next guess is 25. If too low, your next guess is 75.

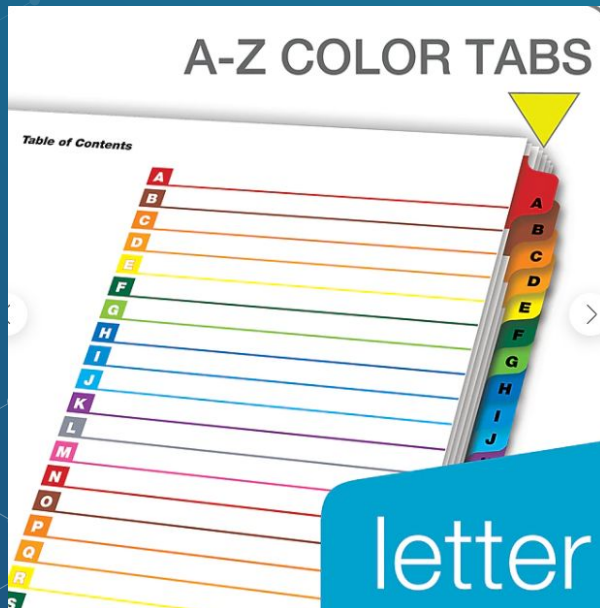
If 25 is too high, we will guess 12. If 25 is too low, we would instead guess 37.

And we continue this approach until we find the number.

This works because the numbers are in an order. (Binary Search)



Indexes



412 INDEX

linear regression, 164–172, 175
 linguistic variables, 339, 347–348, 383
 lower approximation, 186

M
 mammograms, digital, 25–26
 market basket data, 7–8, 29–35, 70, 289
 maximum–minimum mapping, 236
 membership, degree of, 16–17, 340–341, 346, 383
 minimal description length (MDL), 99, 112–113
 minimum confidence (*minconf*), 8, 30, 55
 minimum spanning tree (MST), 292
 minimum support (*minsup*), 8, 30, 55
 monotonic support, 48
 multivariate analysis, 284

N
 natural-language queries, 18–19, 21–22, 22–23, 24–25, 26–27
 neighbor link, 50
 neural networks, artificial, 12–14, 225–228, 267–268
 backpropagation (BPNN), 239–246, 267
 bidirectional associative memory (BAM), 246–250, 267
 feedback networks, 229, 230, 234
 feedforward networks, 229–230, 231, 234
 learning vector quantization (LVQ), 250–255, 267
 probabilistic neural network (PNN), 255–266, 267
 supervised learning, 230–231, 267
 unsupervised learning, 228–230, 267

noisy data, 2, 11–12, 18, 28, 104–105, 127, 314–315
 normalization methods, 236–239
 normal probability distribution, 109, 149
 nuanced information, 346
 null hypothesis, 144–145
 numerical attributes. *See* quantitative attributes

O
 ordinal scale, 287
 output node, 227
 overfitting, 86, 96, 97, 104, 108, 110
 overlapping tuples, 44

P
 partial completeness, 57–58
 partial decision tree, 9, 80, 123–128, 129
 partition algorithms for clustering, 307–310, 329
 post-processing, 4
 post-pruning, 89, 96–97, 108–110, 118–122
 pre-processing, 4, 5–6
 pre-pruning, 104, 108
 Prism algorithm, 83–86, 129
 probabilistic decision rules, 189, 212
 probabilistic mapping, 237–238
 probabilistic neural network (PNN), 255–266, 267
 probability, 62, 86–89
 conditional, 62, 69, 146–148, 175, 182, 183–184, 186, 212
 F-distribution, 168
 hypergeometric, 88
 normal distribution, 109, 149



How does an index help?

An index is a data structure associated with a table or view and stores the values for a specific column and a reference to the row. Furthermore it keeps these in an order so that we can play our "game".

The index data structure is designed so that lookup operations are very fast - usually in logarithmic time: $O(\lg n)$

If our queries utilize fields that have been indexed, we don't need to look through every single row. Instead we only need to look through a much smaller subset.



How does an index help?

We want to execute the same query, but now we can use an index

```
SELECT * FROM Employees WHERE Salary = 38000
```

How do you think it executes your query?

Salary
25000
25000
25000
30000
38000
40000
43000
55000

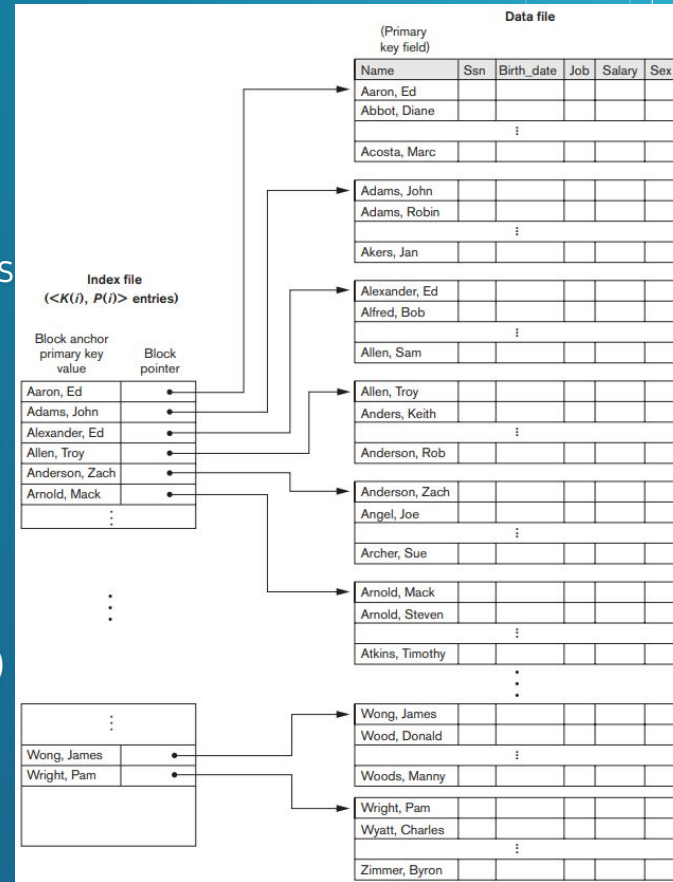
EMPLOYEE									
Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

Primary Index

A primary index is an ordered file whose records are of fixed length with two fields and it acts like an access structure.

The first field is of the same data type as the ordering key field (called the primary key) of the data file, and the second field is a pointer to a disk block

There is one index entry (or index record) in the index file for each block in the data file.





Dense vs Sparse Index

- ◇ Indexes can also be characterized as dense or sparse.
- ◇ A dense index has an index entry for every search key value (and hence every record) in the data file.
- ◇ A sparse (or nondense) index, on the other hand, has index entries for only some of the search values
- ◇ (Thus, a primary index is a sparse index)

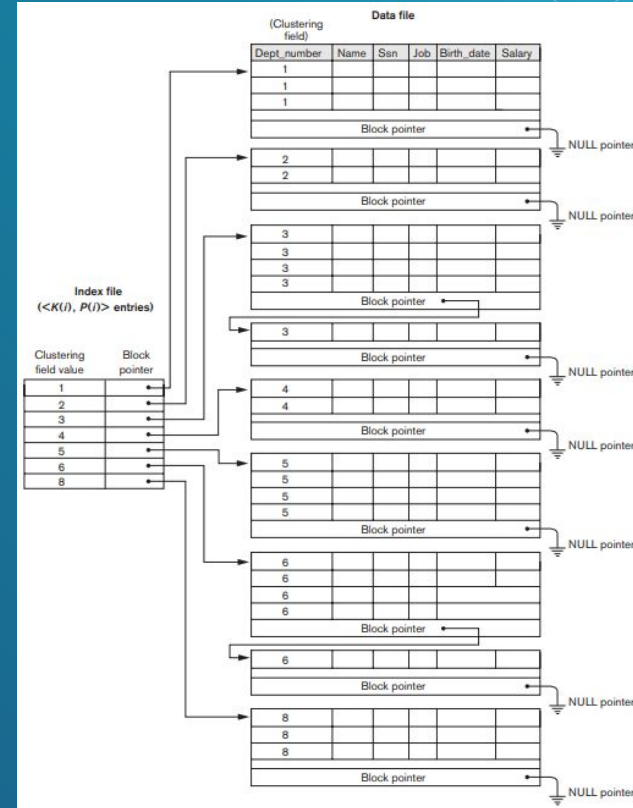
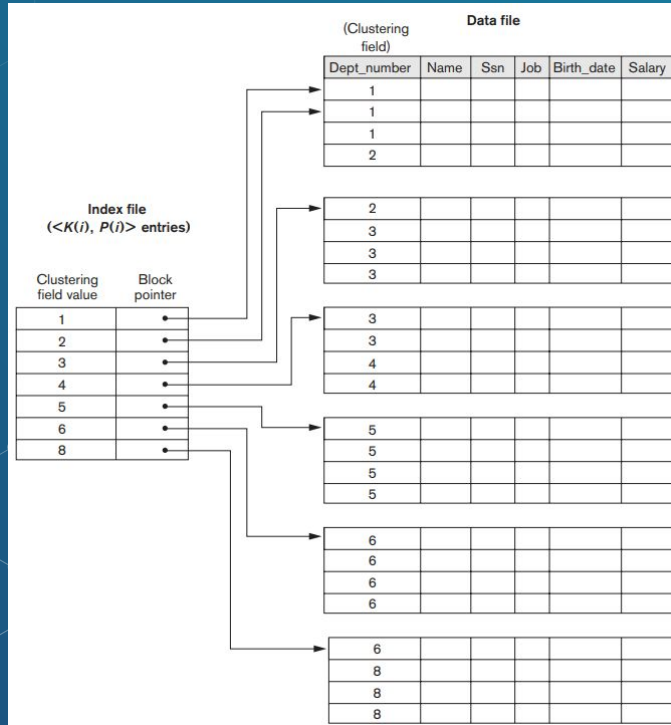


Clustered Index

- ◇ If file records are physically ordered on a nonkey field—which does not have a distinct value for each record—that field is called the clustering field and the data file is called a clustered file.
- ◇ We can create a different type of index, called a clustering index, to speed up retrieval of all the records that have the same value for the clustering field.
- ◇ This differs from a primary index, which requires that the ordering field of the data file have a distinct value for each record.
- ◇ A clustering index is another example of a sparse index because it has an entry for every distinct value of the indexing field.
- ◇ Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, but not both.



Clustered Index





Secondary Indexes



Secondary Index

- ◇ A secondary index provides a secondary means of accessing a data file
- ◇ The data file records could be ordered, unordered, or hashed.
- ◇ The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values.
- ◇ The index is again an ordered file with two fields.
- ◇ The second field is either a block pointer or a record pointer.
- ◇ Many secondary indexes (and hence, indexing fields) can be created



Secondary Index

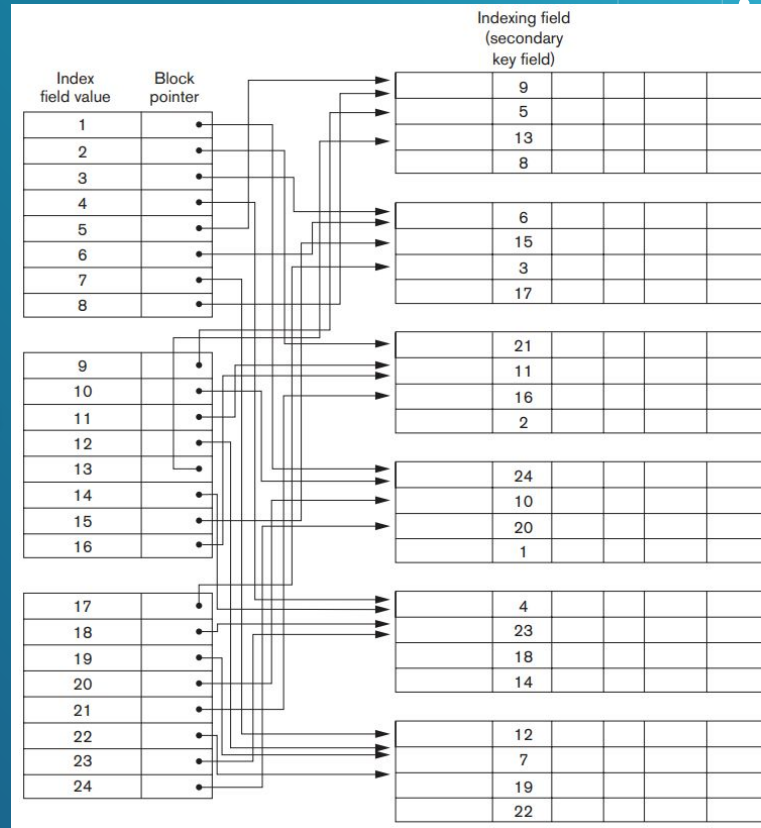
First we consider a secondary index access structure on a key (unique) field that has a distinct value for every record. Such a field is sometimes called a secondary key; in the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. In this case there is one index entry for each record in the data file, which contains the value of the field for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is dense



Secondary Index

Note here this is for a secondary key field, so each value is unique.

Because the records of the data file are not physically ordered by values of the secondary key field, we cannot use block anchors.





Secondary Index

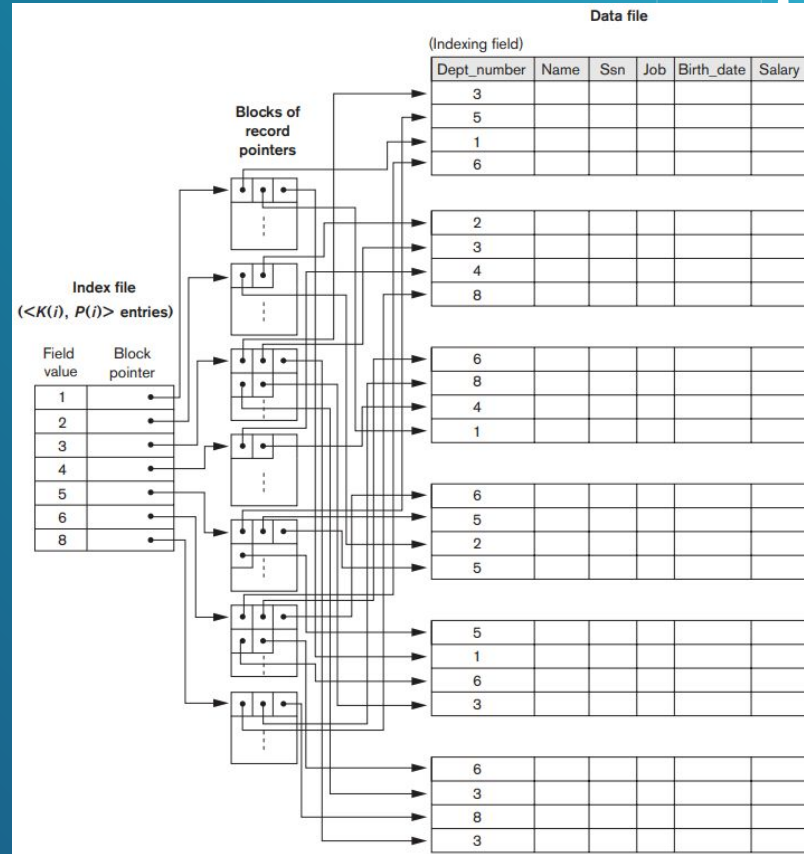
We can also create a secondary index on a nonkey, nonordering field of a file. In this case, numerous records in the data file can have the same value for the indexing field

- ◇ Option 1 - include duplicate index entries with the same key value one for each record.
- ◇ Option 2 - have variable-length records for the index entries, with a repeating field for the pointer
- ◇ Option 3 - keep the index entries themselves at a fixed length and have a single entry for each index field value, but to create an extra level of indirection to handle the multiple pointers. This is the most common choice.



Secondary Index

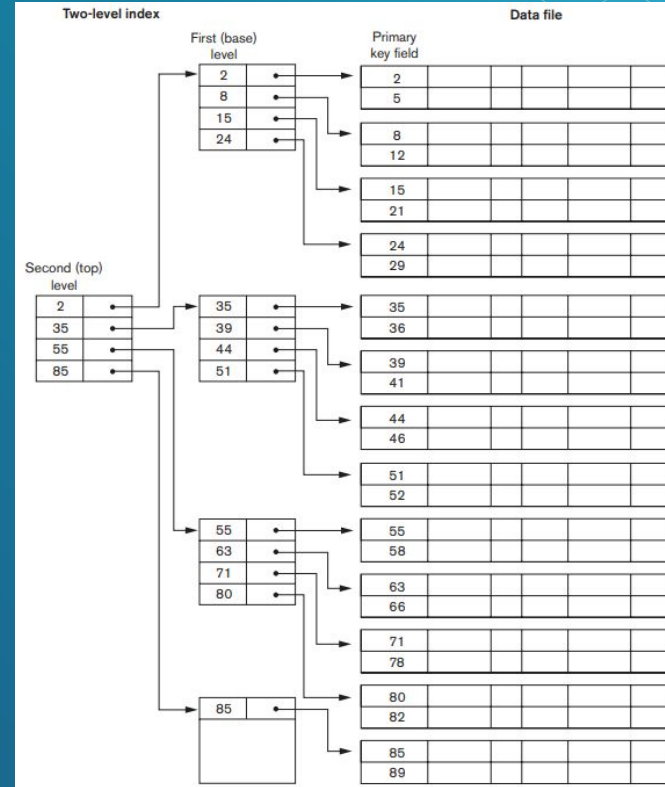
Option 3 Illustrated:



Multilevel Index

Although our last example still had a single index file, it begets the idea of an indirect multi-level approach.

By adding layers, a multilevel index can make use of block anchors to 'chop' up the full index into pieces that are quickly searched.

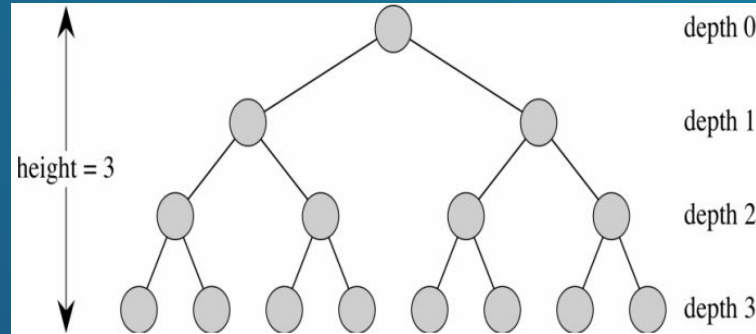




B-Trees and B+ Trees

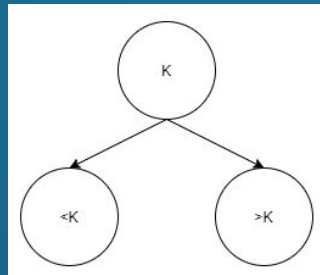
Binary Trees

- The maximum number of nodes of depth k is 2^k .
- A full binary tree of depth k has $2^{k+1}-1$ nodes.
- Degree of a tree is the max number of subtrees of any node (the # of children). Sometimes referred to as k -ary. Binary trees are of degree two.
- Depth is the distance of the node from the root.
- Height is the maximum depth of any node in the tree.



Binary Search Trees

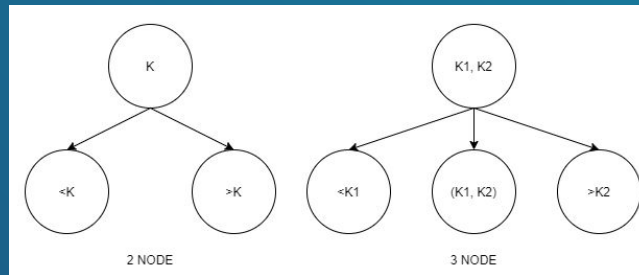
The fundamental property of a binary search tree is that within any given node a key (k) is stored. The left child will contain a key less than k while the right child will contain a key greater than k .



2-3 Trees

Here, every node with children will have either

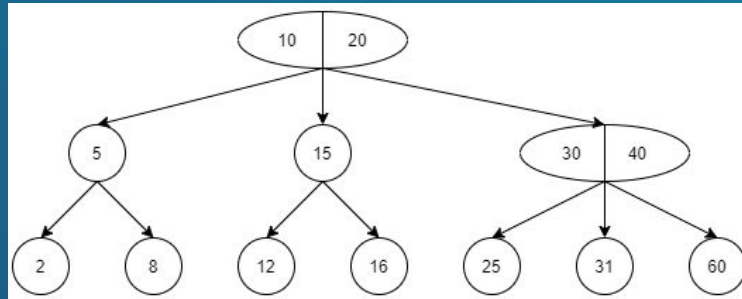
- Two children and one data element or
- Three children and two data elements



2-3 Trees

Find 17: $[10|20] \rightarrow [15] \rightarrow [16]$

Find 22: $[10|20] \rightarrow [30|40] \rightarrow [25]$

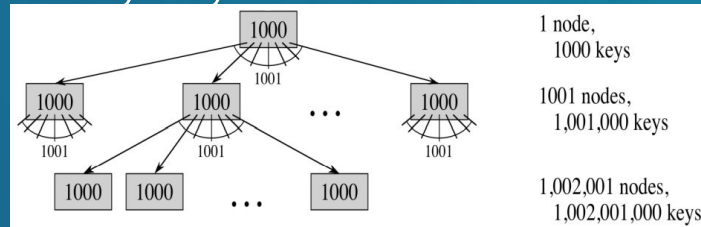


B-Trees

B-trees are balanced search trees designed to work well on disks or other secondary storage devices. (Root is in memory)

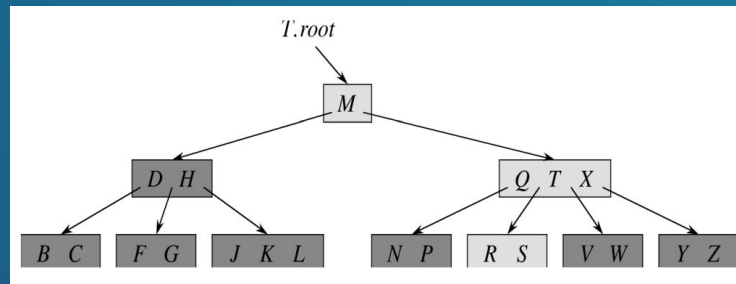
We often see branching factors between 50 and 2000 depending on the size of a key relative to the size of a page.

A B-tree with branching factor of 1001 and height 2 can store over one billion keys. (We can find any key in the tree with at most only two disk accesses)



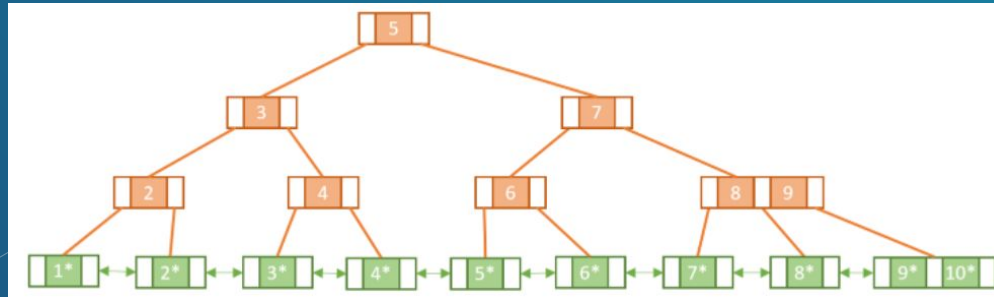
B-Tree Summary

- ◇ All the leaf nodes of the B-tree must be at the same level.
- ◇ Above the leaf nodes of the B-tree, there should be no empty sub-trees.
- ◇ B- tree's height should lie as low as possible.
- ◇ All internal and leaf nodes have data pointers.



B+ Tree Summary

- ◇ A search requires you to travel all the way to a leaf node.
 - ◇ Only leaf nodes have data pointers.
 - ◇ All leaf nodes are linked together in a doubly-linked list.
 - ◇ Most RDBMS use B+ Trees for indexing.
- No internal data, means maximum number of keys can be stored, thus minimizing number of levels.





Hashing



Hash Function

A hash function is a mathematical function that:

- ◇ takes an input value from a set with many (or infinite) members
 - ◆ Example: Modulus operator
- ◇ produces an output from a set with a fixed (fewer) number of members.
- ◇ Hash functions are not reversible.
- ◇ *Should* be quickly computable.
- ◇ *Should* evenly distribute the keys.



Modulus

We can try to use the idea of modulus as this will provide a max number 'cap' in which the numbers will repeat in a cycle:

$$0 \% 6 = 0$$

$$1 \% 6 = 1$$

$$2 \% 6 = 2$$

$$3 \% 6 = 3$$

$$4 \% 6 = 4$$

$$5 \% 6 = 5$$

$$6 \% 6 = 0$$

$$7 \% 6 = 1$$

Any number will 'turn into' something that fits in the range $[0 \dots (m-1)]$



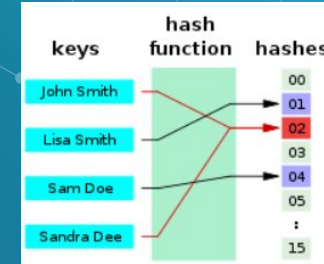
Hash Collisions

A collision is when two different keys share the same hash value.

(i.e. they are put into the same 'bucket')

Due to the pigeonhole principle, if we are taking a large number of keys and putting them into a smaller number of buckets, this is inevitable.

Each key value is hashed to a bucket which then contains pointers to relevant data.



	Emp_id	Lastname	Sex
Bucket 0	13646
	21124

Bucket 1	23402
	81165

Bucket 2	51024
	12676

.....	12676	Marcus	M	..
.....	13646	Hanson	M	..
.....
.....	21124	Dunhill	M	..
.....	23402	Clarke	F	..
.....
.....	34723	Ferragamo	F	..
.....
.....	41301	Zara	F	..
.....
.....	51024	Bass	M	..



Bitmap Index

Bitmap Index

Suppose we have fields that have a small number of unique values.

Here the fields Sex and Zipcode qualify:

A bitmap index is built on one particular value of a particular field and is a bit array.

The i^{th} bit is set to 1 if the row i has that value, and 0 otherwise.

EMPLOYEE					
Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap Index for Sex							
M							
F							
Bitmap Index for Zipcode							
19046							
30022							
94040							




Bitmap Index

Multi-attribute conditions can be combined easily by using bitwise AND or OR operations.

In general, bitmap indexes are efficient in terms of storage space. Consider a file of 1 million rows/records. A bitmap index would only take up 1 million bits, or ~125 Kbytes.

However, when records are deleted, renumbering rows and shifting bits becomes expensive.

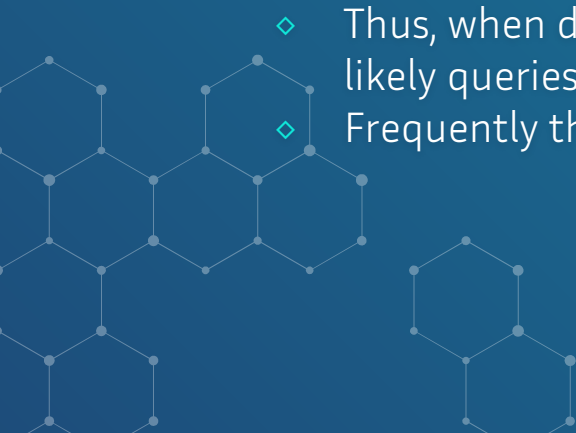





What's the catch?



What's the catch?

- ◇ An index takes up space.
The larger the table, the more space it requires!
 - ◇ An index needs to be updated.
Thus, whenever a row is added, removed, or updated the same operations will need to be performed to the index.
 - ◇ Thus, when designing a database index keep in mind what will be the likely queries.
 - ◇ Frequently the primary key is used.
- 
- 



Should you always use an index?

Indexes are less important when:

- ◇ Small tables
- ◇ Big tables when queries process most/all of the rows

(In short, know your use cases)





Query Examination

Examining Queries

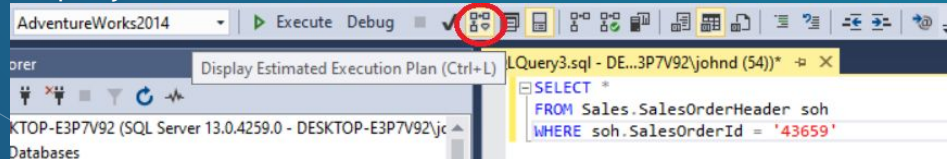
Many DBMS have means to examine how queries will run, but are database dependent. It displays information from the optimizer about the statement execution plan

e.g. how tables are joined and in which order

MySQL: EXPLAIN SELECT * FROM Table

<https://dev.mysql.com/doc/refman/8.0/en/explain.html>

SQL Server: "Display Estimated Execution Plan"





Examining Queries

Benefits:

- ◇ where indexes should be added
- ◇ whether the tables are joined in an optimal order
- ◇ should a query be written differently?

Example

We have a table Person that has an index on the Lastname field.

```
-- query 1  
SELECT LastName  
FROM Person.Person  
WHERE LastName = 'Smith';
```

```
-- query 3  
SELECT LastName  
FROM Person.Person  
WHERE LastName LIKE '%mith';
```

```
-- query 2  
SELECT LastName  
FROM Person.Person  
WHERE LastName LIKE 'Sm%';
```

```
-- query 4  
SELECT ModifiedDate  
FROM Person.Person  
WHERE ModifiedDate BETWEEN '2000-01-01' and '2000-01-31';
```


Example

How does q1 compare to q2?

```
-- query 1  
SELECT LastName  
FROM Person.Person  
WHERE LastName = 'Smith';
```

```
-- query 2  
SELECT LastName  
FROM Person.Person  
WHERE LastName LIKE 'Sm%';
```

Both queries are using the same indexed field.

One is looking for last names that are exactly 'Smith', the other is looking for last names that begin with 'Sm'.

This means they can quickly jump to the 'Sm' entries in the index, and roughly look at the same number of entries.

Example

How does q2 compare to q3?

```
-- query 2  
SELECT LastName  
FROM Person.Person  
WHERE LastName LIKE 'Sm%';
```

```
-- query 3  
SELECT LastName  
FROM Person.Person  
WHERE LastName LIKE '%mith';
```

Query 2 is looking for last names that begin with 'Sm' and can quickly get to just those within the index.

Query 3 is looking for last names that end in 'mith', and in theory any entry in the index could match that. So this query has to go down one by one for every single entry of the index looking for matches.

Example

How does q3 compare to q4?

```
-- query 3  
SELECT LastName  
FROM Person.Person  
WHERE LastName LIKE '%mith';
```

```
-- query 4  
SELECT ModifiedDate  
FROM Person.Person  
WHERE ModifiedDate BETWEEN '2000-01-01' and '2000-01-31';
```

Query 3 as we saw has to look at every entry in the index.

Query 4, however, doesn't have an index. Instead, it has to load up every record within the table and find the data it wants and then perform a comparison. Whereas Query 3 will only have to pull up the records that match.