Indexing

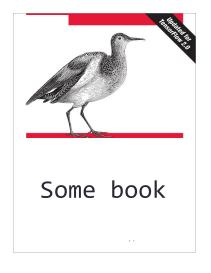
CS 4750 Database Systems

[Silberschatz, Korth, Sudarshan, "Database System Concepts," Ch.14] [H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems: The Complete Book, Ch.14]

© Praphamontripong

Indexing in a Book

Find information about "Google Cloud SQL" from



- Look for the keywords in the index
- Find the pages where the words occur
- Read the pages to find the information

role in data-driven decisions, 4-6 support from cloud computing, 6-10 data exploration (see interactive data exploration) data pipeline batch input and output, 336-338 batching requests, 340 building with Apache Beam/Cloud Dataflow, 101-113, 254-289 data processing portion, 338 debugging, 269, 277-288, 339 real-time stream processing, 119-130, 342-352 running pipelines in the cloud, 109-113, 261.275 source code for, x specifying input and output files, 260 windowing pipelines, 120, 253-288 data processing architectures choices for large datasets, 29-31 choosing between, 38 data in situ. 35-37 scaling out, 33-34 scaling up, 31-33 data science models benefits of Google Cloud Platform for, x building initial, 78-81 continuous training of, 361 creating more complex, 292-295 dangers of overfitting, 292 deep neural network model, 309-312 deploying, 325-329 developing, 166-172 evaluating, 172-179, 212, 232, 361-370 goals of, 132 improving, 308-325 linear classifier, 301-329 logistic regression, 218-235 MapReduce model, 33, 181-190 probabilistic decision making, 13-19 probabilistic decision-making, 168, 191 two-variable Bayesian, 181-215 wide-and-deep models, 314-317 data streaming benefits of, 97 event feeds, designing, 97-99 event streams, publishing, 113-119 real-time stream processing, 119-130, 342-352

388 | Index

using Apache Beam/Cloud Dataflow, 101-113 Data Studio adding BigQuery data source, 128 alternate programs, 81 bar charts, 91-93 benefits of, 82 chart creation, 84-86 Create New Metric option, 86 data source set up, 82 end-user controls, 86-88 explaining contingency tables, 93-95 key concepts in, 82 pie-charts, 88-91 data, filtering defined, 257 on occurrence frequency, 165 using Apache Beam, 257-260 using Apache Pig, 208-212 data, ingesting Airline On-time Performance Data, 23-29 automating, 38-48 data processing architectures, 29-37 scheduling monthly downloads, 48-61 data-driven decisions role of data engineers in, 4-6 support for, 1-4 Dataflow (see Cloud Dataflow) Datalab (see Cloud Datalab) datalab.bigquery package, 157 Dataproc (see Cloud Dataproc) debugging, 269-271 decision hyperplanes, 330 decision thresholds, 80 decisions, making better based on data, 1-22 deep neural networks, 309-312 dimensionality reduction, 250 dimensionality, curse of, 217 directed acyclic graph (DAG), 187 distributed training defined, 291 in the cloud, 307 process of, 298 DoFn interface, 259 downloads, scheduling, 48-61 dplyr package, 134 duplicates, removing, 285-288 dynamic resource allocation 36

Eclipse, 256, 259, 270 embeddings, 250, 312-314 empirical probability distribution function, 169 end-user controls, adding in Data Studio, 86-88 evaluating model performance, 361-370 evaluation pipeline, 362 event feeds building batches in Cloud Pub/Sub, 117 creating in Apache Beam, 108 designing, 97-99 publishing batches in Cloud Pub/Sub, 118 publishing event streams, 113-119 eventual consistency, 46 Experiment class (TensorFlow), 299-308, 329 exploratory data analysis (EDA) benefits of dashboard creation for, 68 challenges of, 133 vs. dashboard creation, 65, 133 development of, 131 goals of, 132 importance of, 65 exponential distribution, 347 Extract-Transform-Load (ETL) pipeline, 99 FarmHash library, 174 feature crossing, 314 feature engineering, 235-251 categorical variables, 248 creating held-out dataset, 238 defined, 236 dimensionality reduction, 250 feature selection, 239 feature transforms, 244 framework for, 236 scaling and clipping features, 242 value of simpler models, 235 feature selection, 236, 239

federated queries, 142-144 filtering data (see data, filtering) Flask, 57 Flume, 101 frequency analysis attacks, 376

gcloud command, 73, 184 geographical maps, 69

Google App Engine, 49 Google BigQuery (see BigQuery) Google Cloud Client library, 74 Google Cloud Platform (GCP) benefits of, x bisection speeds on, 188 creating Cloud SQL instances, 72 effect on data science, 10-12 ingesting data into the cloud, 23-62 interacting with, 73 monitoring API usage, 339 persistent drives in, 32 preemptible machines on, 203 project creation, 20-22 reference architecture, 119 REST API actions, 74 upport for data engineers, o Google Cloud SOL

creating instances, 72 loading data onto, 71 streaming pipelines and, 33 e Cloud Storage file handling in, 46

job speeds on, 35 removing duplicates, 285-288 staging BigQuery data on, 136 uploading data to, 45-48 Google File System (GES) 36 184 Graphic Detail blog, 70 graphics, types of, 69 (see also dashboards)

Hadoop Distributed File System (HDFS), 33, 36, 184 Hadoop ecosystem, 187 head-of-line blocking, 37 --help command, 73 hexbin plot, 193 histogram equalization, 198-202

hyperparameter tuning, 317-325, 330 independence check, 193-195 independent samples, 172 ingesting data into the cloud, 23-29 initialization actions, 189

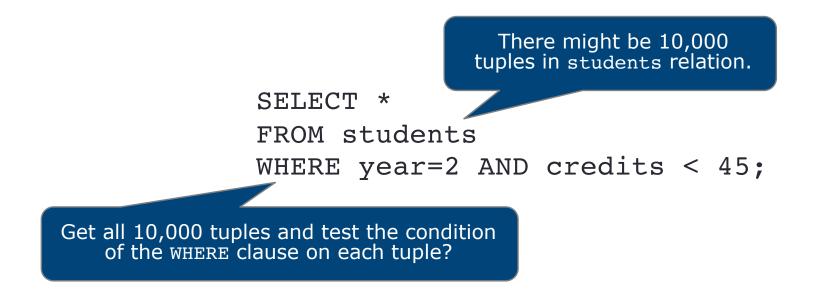
Index | 389



G

Indexing in Database

Example: Find 2nd year students who have taken < 45 credits



- Find which disk block the corresponding record resides
- Fetch the disk block
- Get the appropriate student records

Is there a way to get only the tuples from 2nd year students and then test each of them to see if the credits match?

Indexing in Database

- Indexing = data structure technique to optimize the performance of a database by minimizing the number of disk accesses required when query is processed
- Basic algorithm to search linear. However, complex search queries (especially with joins) impacts performance
- Indexing helps improve performance

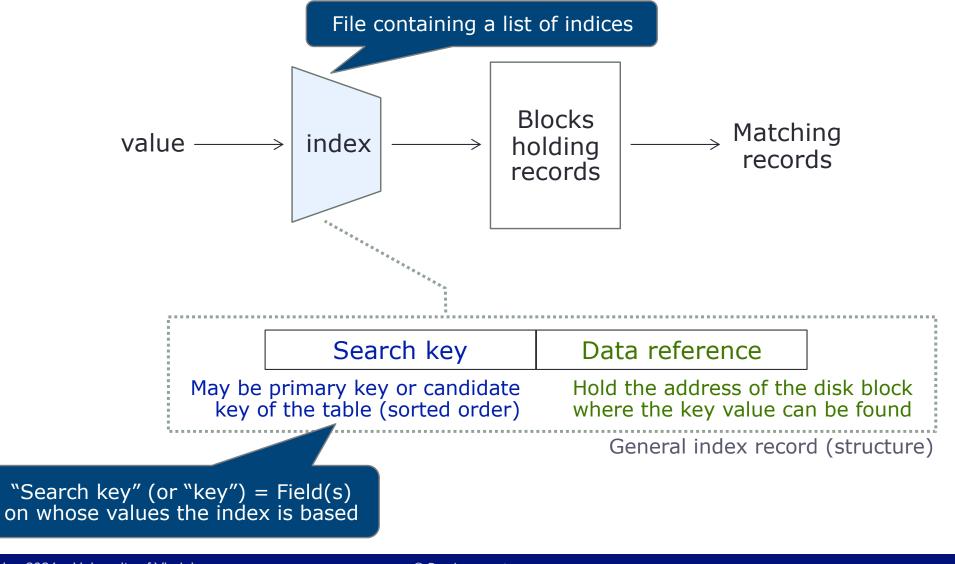
Common structure used by a typical DBMS is B+ tree

The key for the index can be any attribute or set of attributes and need not be the key for the relation on which the index is built.

Rule of thumb: Create an index on the attribute that is used frequently in the search

Indexing in Database

 An index takes a value for some field(s) and finds records with the matching value quickly



Selection of Indexes

- An index on an attribute may speed up the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute.
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

Database designers must analyze the trade-off

Which Indexing Technique to Use

Aspects that must be considered:

- Access types
 - Finding records with a specified attribute value (search key), or
 - Finding records with attribute value based on a specified range
- Access time
 - Time needed to find a particular data item or set of data items
- Insertion time
 - Time to find the place to insert + time to insert a new data item + time to update the index structure
- Deletion time
 - Time to find the data item to be deleted + time to delete the data + time to update the index structure
- Space overhead
 - Space occupied by an index structure (vs. performance)

Types of File Organization Mechanism

Sequential file organization (or Ordered index)

- Indices based on sorted ordering of the values
- Generally fast
- Basic / traditional structure that most DBs use

Hash file organization (or Hash index)

- Indices based on a uniform distribution of values across a range of buckets
- Hash function determines a value assigned to a bucket

Example: Sequential File

Instructor

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	$ \prec $
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
		0		

Instructor records

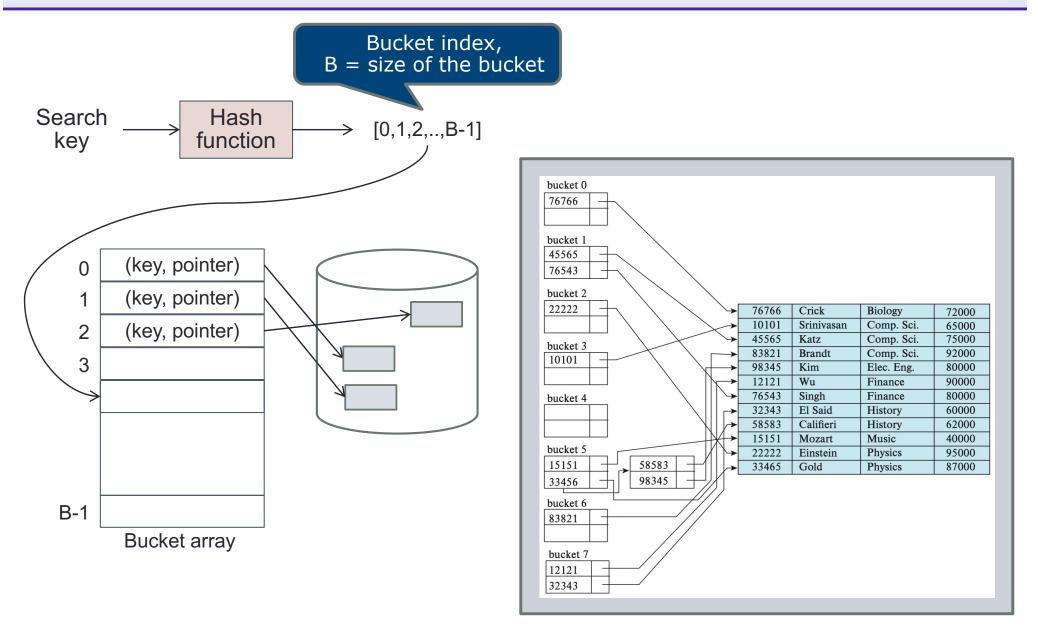
 The records are stored in sorted order of instructor's ID (used as a search key)

 Search key defines the sequential order of the file

[Ref: Figure 11.1, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 477]

© Praphamontripong

Example: Hash File



[Ref: based in part on Figure 24.6, Silberschatz, Korth, Sudarshan, "Database System Concepts," 7th Ed., page 1191]

Ordered Index Structures

A file may have several indices, on different search keys

Primary index (or clustered index)

- Search key defines the sequential order of the file
- Search key of a clustering index is often the primary key (but not necessarily so)

Secondary index (or unclustered index)

- Search key specifies an order different from the sequential order of the file
- Use an extra-level of indirection to implement a secondary index, containing pointers to all the records

Example: Primary Index

Search key defines the sequential order of the file

Fast but can result in unnecessary indices and big space needed

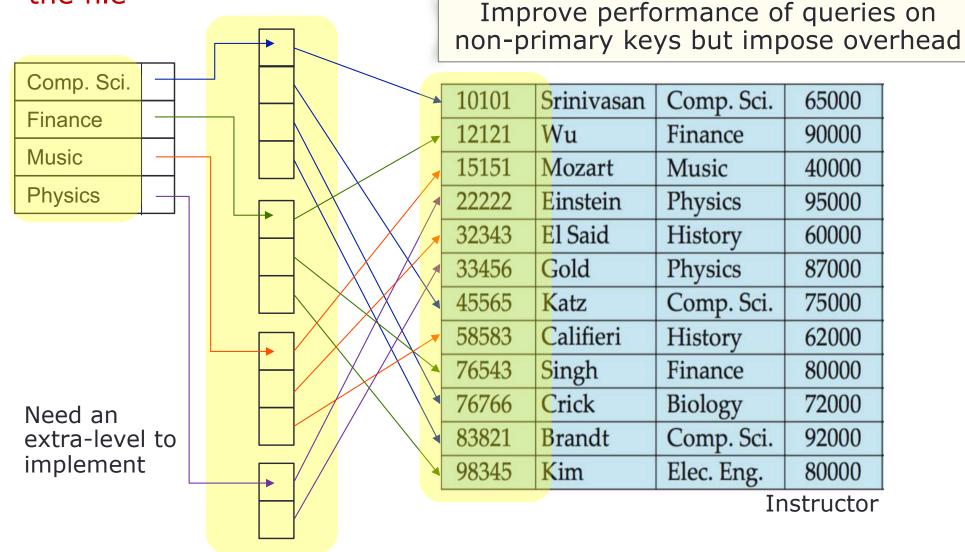
		1			-		
10101		, ,	10101	Srinivasan	Comp. Sci.	65000	
12121		,	12121	Wu	Finance	90000	\sim
15151	6	, ,	15151	Mozart	Music	40000	\prec
22222	-	, ,	22222	Einstein	Physics	95000	\prec
32343	-	,	32343	El Said	History	60000	\prec
33456	-	, ,	33456	Gold	Physics	87000	K
45565		,	45565	Katz	Comp. Sci.	75000	K
58583] ,	58583	Califieri	History	62000	K
76543	1],	76543	Singh	Finance	80000	\prec
76766		<mark>]</mark> ,	76766	Crick	Biology	72000	
83821		<mark> </mark> ,	83821	Brandt	Comp. Sci.	92000	K
98345		,	98345	Kim	Elec. Eng.	80000	
6.5		71	8 ⁴ 13				10 234

Instructor

[Ref: Figure 11.2, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 478]

Example: Secondary Index

Search key specifies an order different from the sequential order of the file



[based in part on Figure 11.6, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 484]

Ordered Index

- Created on the basis of the key of the table
- Ordered file with fixed, two fields

	Search key	Data reference
Ca	May be primary key or andidate key of the table (sorted order)	Hold the address of the disk block where the key value can be found

- Unique to each record (i.e., 1:1 mapping)
- Since primary keys are stored in sorted order, the performance of the search operation is quite efficient
- Two types:
 - Dense index
 - Sparse index

Dense Index

- A record is created for every search key value
- Need more space to store index records
- Example: a search key is a primary key

Find instructor with ID "58583"

Instructor

10101 Srinivasan Comp. Sci. 65000 10101 A primary index 12121 Wu Finance 90000 12121 that is dense on a 40000 15151 Mozart Music 15151 primary-key, 22222 Einstein 95000 22222 Physics ordered table is 32343 32343 El Said History 60000 redundant Gold Physics 87000 33456 33456 (requiring 45565 Katz Comp. Sci. 75000 45565 unnecessary Califieri 58583 58583 History 62000 spaces) 76543 76543 Finance 80000 Singh 76766 Crick Biology 72000 76766 Brandt 92000 83821 83821 Comp. Sci. 98345 98345 Kim Elec. Eng. 80000 Instructor file is sorted by instructor's ID Point to the real record on the disk (every search key valued)

[Ref: Figure 11.2, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 478]

© Praphamontripong

Dense Index

- Support range queries
- Example: a search key is not a primary key

Find a history instructor with ID "58583"

Pointer points to the first data record with the search-value. The rest of the records are sorted on the same search key

	Biology	→	76766	Crick	Biology	72000	
	Comp. Sci.		10101	Srinivasan	Comp. Sci.	65000	
	Elec. Eng.		45565	Katz	Comp. Sci.	75000	
	Finance		83821	Brandt	Comp. Sci.	92000	
	History		98345	Kim	Elec. Eng.	80000	
	Music		12121	Wu	Finance	90000	
	Physics		76543	Singh	Finance	80000	
Instructor file is sorted on the search key <i>dept-name</i>		32343	El Said	History	60000		
		58583	Califieri	History	62000		
		$\langle \rangle$	15151	Mozart	Music	40000	
		22222	Einstein	Physics	95000		
			33465	Gold	Physics	87000	
Follow the pointer directly to the first record. Instructor $-$					uctor 🖵		

Follow the pointer directly to the first record, then follow the pointer in that record to locate the next record in search key order until the desired record is found

[Ref: Figure 11.4, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 480]

© Praphamontripong

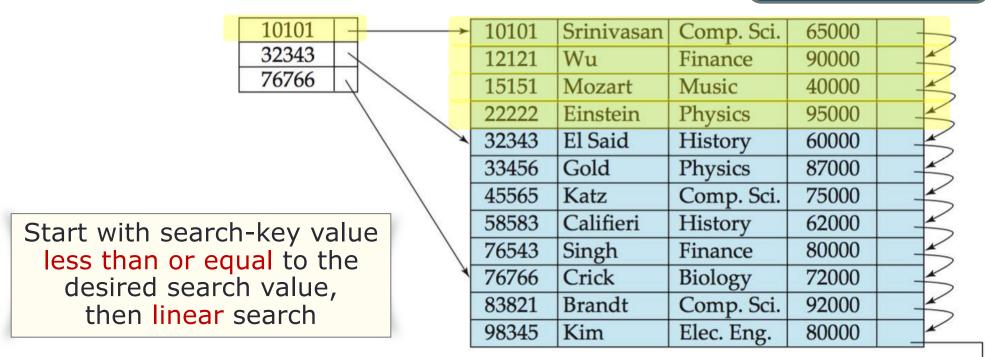
Dense Index: Lookup

- Given a search key *K*, the index is scanned
 - When K is found, the associated pointer to the data file recorded is followed and the block containing the record is read in main memory
- When dense indexes are used for non-primary key, the minimum value is located first
 - Consecutive blocks are loaded in main memory until a search key greater than the maximum value is found
- The index is usually kept in main memory. Thus one disk I/O has to be performed during lookup
- Since the index is sorted, a binary search can be used.
 - If there are n search keys, at most log₂n steps are required to locate a given search key
- Query-answering using dense indices is efficient

Sparse Index

- Used when dense indices are too large
- One key-pointer pair per data block
- Can be used only if the relation is stored in sorted order of the search key

Find instructor with ID "22222"



Instructor

[Ref: Figure 11.3, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 479]

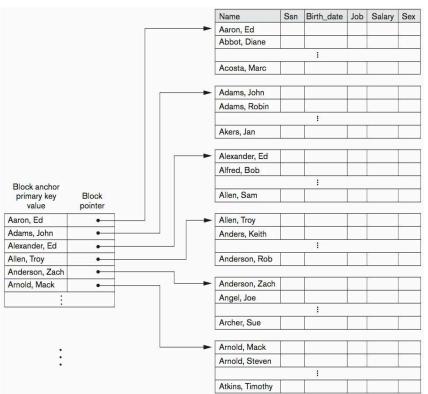
Sparse Index: Lookup

- Given a search key K,
 - Search the sparse index for the greatest key $\leq K$, using binary search
 - Retrieve the pointed block to main memory to look for the record with search key K (linear search vs. binary search)
- The index is usually kept in main memory. Thus one disk I/O has to be performed during lookup
- Efficient in space but may require more computation time due two binary searches
 - Search on the sparse index
 - Search on the retrieved data block

Dense Index vs. Sparse Index

- A primary index that is dense on an ordered table is redundant
- Thus, a primary index on an ordered table is always sparse
- Dense indices are faster in general
- Sparse indices require less space and impose less maintenance for insertions and deletions
- Try to have a sparse index with one entry per block





[Ref: Figure 18.1, Elmasri Navathe, "Fundamentals of Database Systems," 6th Ed., page 634]

More Info on Indexing

- When a primary key is created for a table, a table is ordered based on the primary key
- At least one sparse index is created on that record to reduce search time
- If a column (or some columns) is declared as unique, a secondary index is created
- Every index introduces more data and more overhead (especially when doing insert, delete, or update)

Can we create (or add) indices to just any table (or DB)? – **No!**

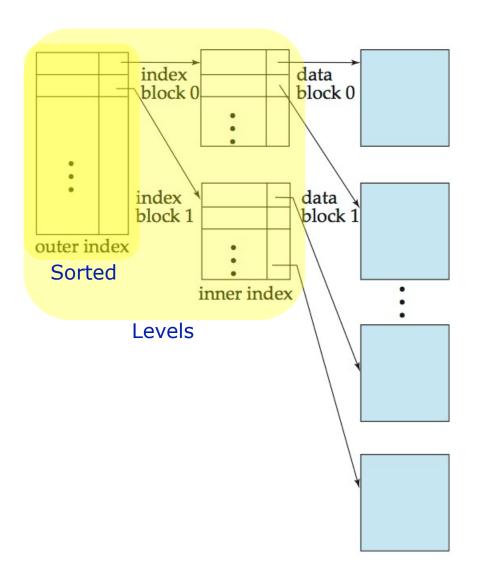
Read-heavy DBs – can index a lot (if space allows) Write-heavy DBs – index sparingly (take a balanced approach) Write-ONLY DBs – one or no index

Multi-Level Indices

- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low
- If index is too large to be kept in main memory, index blocks must be fetched from disk when required. One search results in several disk-block reads
 - If no overflow blocks in the index \rightarrow use binary search
 - If overflow blocks \rightarrow use sequential search
- Solution:
 - Use a sparse index on the index

Example: Two-Level Sparse Index

- Use binary search on outer index
- Scan index block until the correct record is found
- Scan block pointed to for desired record
- For very large files, add additional level of indexing to improve search performance
- Must update indices at all levels when perform insertion or deletion



[Ref: Figure 11.5, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 481]

Updating Indices

All associated indices must be updated when a record is inserted into or deleted from a file

Insertion:

- Find a place to insert
- For dense index:
 - Insert search key value if not present
- For sparse index:
 - No change unless a new block is created
 - If the first search key value appears in the new block, insert the search key value into the index

Deletion:

- Find the record
- If it is the last record, delete that search key value from index
- For dense index:
 - Delete the search key value
- For sparse index:
 - Delete the search key value
 - Replace the key value's entry index with the next search key value if not already present

How to Implement: Use B+ Trees

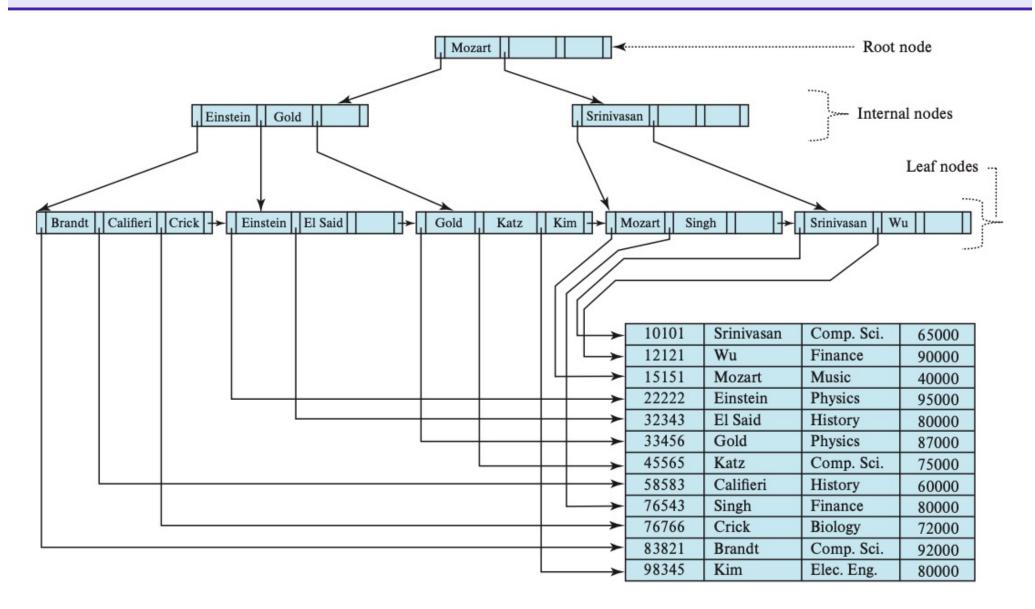
- As the database grows (and the index file grows), performance degrades. Reorganization is costly
- Solution: use B+ tree to maintain indices

B+ Tree

- A tree-like file structure
 - Links nodes with pointers
 - Has exactly one root, bounded by leaves
 - Has unique path from root to each leaf; all paths are equal length
 - Store keys only at leaves, references in other/internal nodes
 - Guides key search via the reference values, from root to leaves
- Balance same length on every path from root to leaves
- Extensible number of pointers (n) at any given node

Balance sorted tree, allowing fast search and maintenance without overflow pages

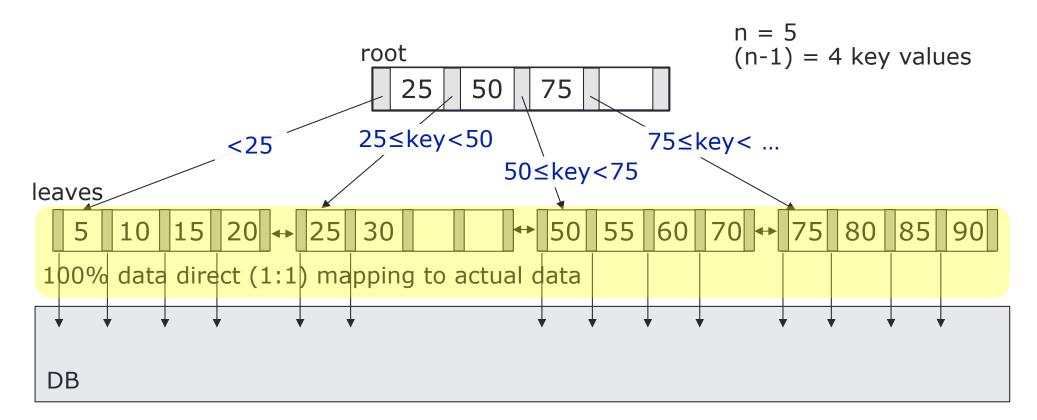
Example: B+ Tree



[Ref: Figure 14.9, Silberschatz, Korth, Sudarshan, "Database System Concepts," 7th Ed., page 636]

B+ Tree: Node

- Nodes: root, internal node, or leaf node
- Each node is exactly one disk page ("page" and "node" are used interchangeably)
- Contain n pointers and (n-1) key values



B+ Tree: Leaf Node

- Each entry consists of a key value and a pointer to the storage location of data matching the key
- Leaf nodes are organized into a linked list pages, chaining the leaf nodes



- For a B+ tree with n pointers
 - A leaf node holds $\left[\frac{(n-1)}{2}\right] \le \text{key values} \le (n-1)$
- Example:
 - If n=4, each leaf contains at least $\left[\frac{(4-1)}{2}\right] = 2$ key values and at most (4-1) = 3 key values

[Ref: Silberschatz, Korth, Sudarshan, "Database System Concepts," 7th Ed., page 635, constraints on leaf nodes]

B+ Tree: Internal Node

- Nonleaf nodes: a multilevel (sparse) index
- Each entry consists of a reference value (key) and a pointer to the leaf nodes

- For a B+ tree with n pointers
 - An internal node holds $\left[\frac{n}{2}\right] \le$ pointers $\le \mathbf{n}$ (thus, hold up to (n-1) key values)
- Example:
 - If n=4, each internal node contains at least $\left|\frac{4}{2}\right| = 2$ pointers and at most = 4 pointers

[Ref: Silberschatz, Korth, Sudarshan, "Database System Concepts," 7th Ed., page 635, constraints on nonleaf nodes]

B+ Tree: Root Node

- A root node consists of one or more reference values (keys) and pointers to the leaf nodes (or internal nodes)
- For a B+ tree with **n** pointers
 - A root node holds $1 \leq \text{key values} \leq (n-1)$
 - Must have at least two pointers if the root points to internal nodes
 - Must have at least one entry if the root is the only node in the tree
- Example:
 - If n=4, a root node can hold fewer than $\left|\frac{4}{2}\right| = 2$ pointers
 - It must hold at least 2 pointers (unless the tree consists of only one node)

[Ref: Silberschatz, Korth, Sudarshan, "Database System Concepts," 7th Ed., page 636, constraints on root nodes]

B+ Tree: # Indices

- For a n-order B+ tree with a height of h
 - The maximum number of records indexed is $r_{max} = n^h n^{(h-1)}$
 - The minimum number of records indexed is $r_{min} = 2 \left[\frac{n}{2}\right]^{(h-1)}$

Low n vs. High n

- Small value for n -- Tall and thin B+ tree
 - Advantage: Good consistent performance
 - Equal depth of tree \rightarrow constant lookup time
 - Disadvantage: High overhead when insert/delete
 - Need to reorganize up and down the tree
- Large value for n -- Short and wide B+ tree
 - Advantage: Low overhead
 - Disadvantage: Performance varies

Read-heavy DB \rightarrow consistent time \rightarrow low n

Read/Write-mix DB \rightarrow less overhead \rightarrow high n

Advantages and Disadvantages

Index-sequential files:

Disadvantage:

- Performance degrades as sequential file grows because many overflow blocks are created
- Periodic reorganization of entire file is required

B+ Tree indices are alternatives to index sequential files

B+ Trees are used extensively in all DBMS

B+ Tree index file:

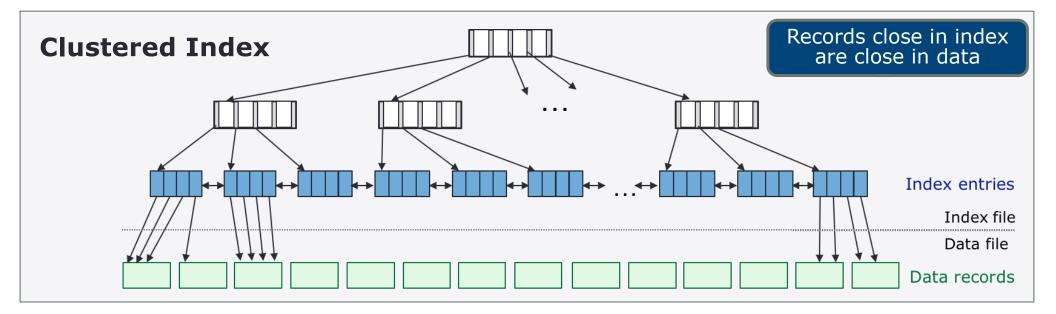
Advantage:

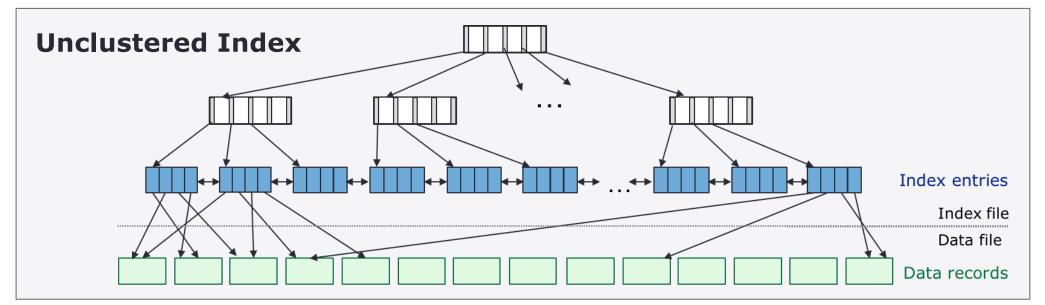
- Automatically recognize itself with small, local changes (when insert or delete data)
- Range queries on indexed attributes are fast

Disadvantage:

• Extra insertion deletion overhead, space overhead

Clustered vs. Unclustered Index





Cost of Disk I/O Operations

Assume a disk block holds 4 tuples of a relation. To find tuples associated with key values 40 - 85

Without an index, need sequential scanning

Estimated cost = # blocks scanned



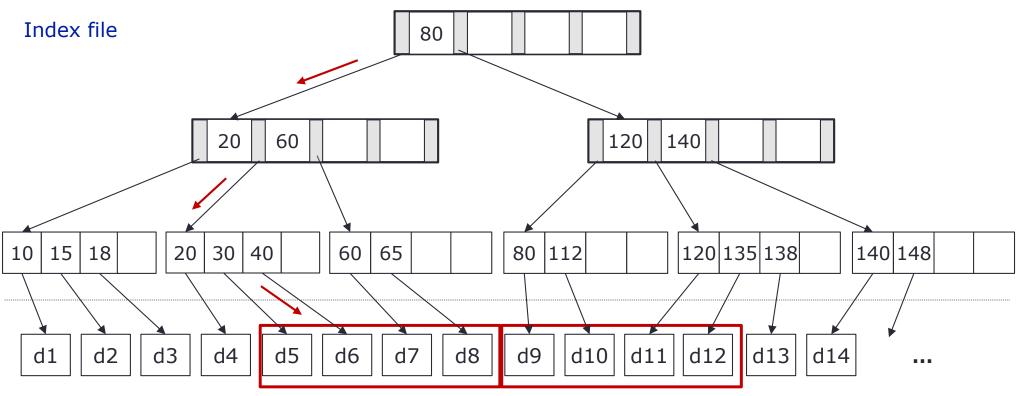
Sequential data file

Cost of Disk I/O Operations

Assume a disk block holds 4 tuples of a relation. To find tuples associated with key values 40 - 85

Use clustered index, index and data are **sorted the same way**

Estimated cost = selectivity estimate x #blocks



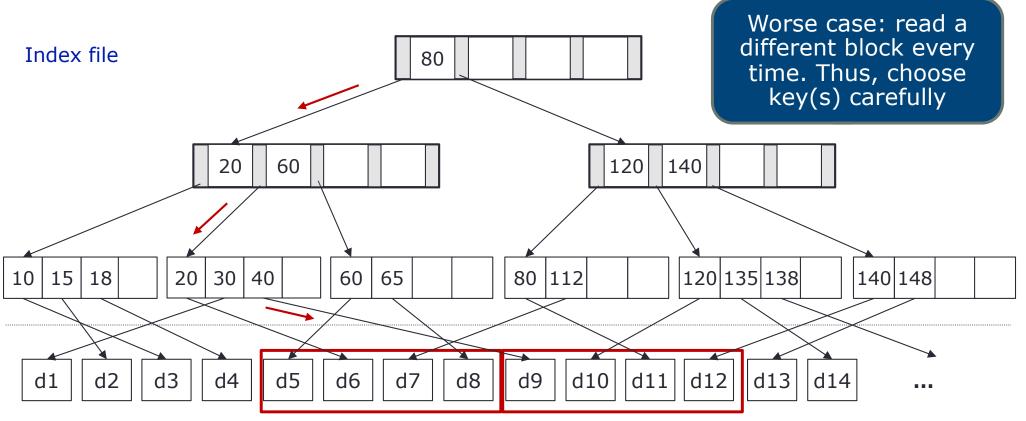
Sequential data file

Cost of Disk I/O Operations

Assume a disk block holds 4 tuples of a relation. To find tuples associated with key values 40 - 85

Use unclustered index, index and data are sorted differently

Estimated cost = selectivity estimate x #tuples



Sequential data file

Another Scenario

Supposed we know the following #blocks = 100 #tuples = 10000

Using an unclustered index in the wrong scenario can lead to low performance

X = Selectivity estimate (~proportion of tuples matching the selection) of a given query = 0.1

No index (full sequential scan)

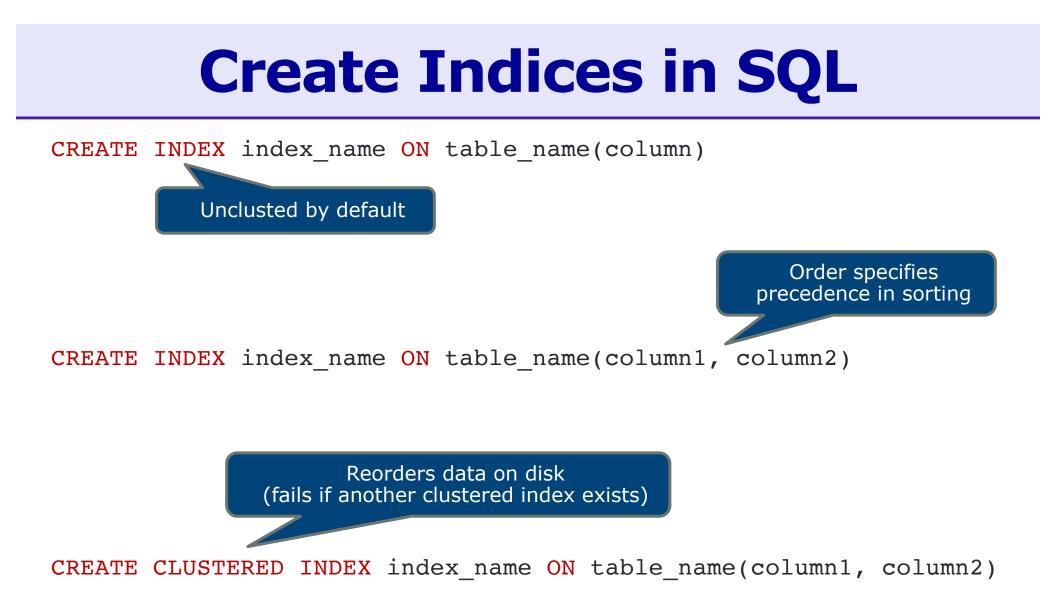
Cost = #blocks = 100

Unclustered index

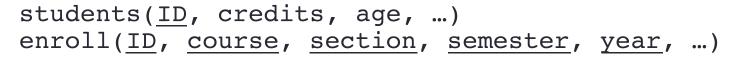
Cost = X * #tuples = 0.1 * 10000 = 1000

Full sequential scan is a better option when

- Selectivity is high (many tuples match the selection), or
- Ratio between #tuples: #blocks is high



Leveraging Indices



What indices could we make on students?

IDs are unique. Unclustered index would do fine.

Expecting 1000 executions/day

Expecting 1000 executions/day

WHERE students.ID = enroll.ID

SELECT * FROM students WHERE credits > 100

FROM students, enroll

Expecting 10 executions/day SELECT *

FROM students

WHERE age > 21

This range query would benefit from a clustered index on credits



This range query would benefit from a clustered index on age

SELECT *

Leveraging Indices (2)

<pre>students(<u>ID</u>, credits, age,) enroll(<u>ID</u>, <u>course</u>, <u>section</u>, <u>sem</u></pre>	Things to consider:
Expecting 1000 executions/day	Size of the expected resultExecution time
SELECT * FROM students, enroll	Without more info, default to clustering on the index that will be used more
WHERE students.ID = enroll.ID	(thus, clustered index on credits)

Expecting 1000 executions/day

SELECT * FROM students WHERE credits > 100

Expecting 10 executions/day SELECT *

FROM students

WHERE age > 21

This range query would benefit from a clustered index on credits



This range query would benefit from a clustered index on age

Wrap-Up

- The existence of an index on an attribute
 - May speed up the execution of the queries (in which a values or a range of values is specified for that attributes), and
 - May speed up joins involving that attribute.
- Every index built for one or more attributes of some relation makes insertions / deletions / updates to that relation more complex and time consuming.
- When creating indices, transactions that will be executed must be taken into account.

Read-heavy DBs – can index a lot (if space allows) Write-heavy DBs – index sparingly (take a balanced approach) Write-only DBs – one or no index

Create indices to match expected query workload