Database Technology

Intermezzo:
Complexity Theory in a Nutshell

Heiko Paulheim
Why?

• Complexity theory
  – essential means of analysis in computer science
  – describes the behavior of an algorithm
  – often not known to non-computer scientists

• Or: what the hell does $O(N^2)$ mean?
What?

• Measure the performance of algorithms
  – how much time does it need? → time complexity
  – how much memory does it need? → memory complexity

• It’s not about *absolute* numbers
  – that would be: it takes 21 seconds

• It’s about *relative* numbers
  – relative to, e.g., no. of rows

• It’s about *scaling*
  – i.e.: what happens if I double the number of rows?
First Example

• Reading N customer records from disk
  – N is a variable
  – each record takes a time t
    • i.e., the total time is N*t

• t may vary
  – e.g., by buying a hard disk twice as fast
  – thus, we usually do not consider t
  – we say: the complexity of reading N customers is O(N)

• O(N) ↔ linear scaling
  – i.e., double the number of customers, double the time
  – the actual hard disk speed does not matter here → O(0.5*N) = O(N)
Second Example

- Storing the pairwise distances between $N$ cities
  - we need to store $0.5*N*N$ distances
  - each distance needs $b$ bytes $\rightarrow 0.5bN$

- Again
  - we may tweak the constant factor $b$
  - e.g., using more/less decimal digits
  - we already know that constant factors do not change the complexity

- $O(N^2)$ $\leftrightarrow$ quadratic complexity
  - twice as many cities $\rightarrow$ four times as many distances to store
  - that is not affected by $0.5$ nor by $b$!
“Calculating” with Complexities

• Constant factors are neglected
  – $O(N) = O(2N) = O(1,000N)$

• The highest complexity dominates the overall complexity
  – $O(N + N^2) = O(N^2)$

• $O(1)$ denotes constant complexity
  – i.e., it is independent of problem size
  – e.g.: add a new record to a table
    • in theory, that should take an equal amount of time
    • irrelevant of the size of the table
Further Notes

• There might be more than one variable
  – e.g., storing a table with N records and C columns uses $O(N \times C)$ memory

• Complexity depends on the solution, not the problem
  – example: storing who is sitting in which office

<table>
<thead>
<tr>
<th>Person</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>B0.01</td>
</tr>
<tr>
<td>Mary</td>
<td>B0.04</td>
</tr>
<tr>
<td>John</td>
<td>B0.02</td>
</tr>
<tr>
<td>Julia</td>
<td>B0.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person</th>
<th>B0.01</th>
<th>B0.02</th>
<th>B0.03</th>
<th>B0.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Mary</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>John</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Julia</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

• Storage and time complexity may be different
  – sometimes, we have to trade them off against each other
Comparison of Complexities

- Complexities can be compared
  - $O(1) < O(\log n) < O(n) < O(n \cdot \log n) < O(n^2) < O(n^c) < O(c^n)$
- Complexity helps analyzing scalability
  - e.g., assessing suitability for larger problems
  - e.g., choosing between different variants
Complexity and Worst Case Behavior

• Complexity describes the worst case behavior
  – think: what happens for very big data?
  – think: what happens in very degraded cases?

• Example for big scales
  – Approach A takes 0.00001*N², approach B takes 10,000*N
  – Unless your N gets very large, you will use A, although O(N²)>O(N)

• Example for degraded cases
  – Storing the ratings of C customers and I items is O(C*I)
  – However, the actual number is much lower
  – Each customer only rates a very small fraction of C
Questions?
Database Technology
Indexing and Hashing

Heiko Paulheim
Previously on Database Technology

- We can find information in databases
  - e.g., employees by name:
    SELECT * FROM employee WHERE name = 'Brandt'
  - e.g., employees within a range of salary
    SELECT * FROM employee WHERE salary > 50000
Finding Information in Databases

• How does that work, actually?
  – SELECT * FROM employee WHERE name = ‘Brandt’

• Naive approach (called *linear search*):
  – Go through the table from top to bottom
  – Find and return all employees with name ‘Brandt’

• Complexity: O(N)
  – Note that even if we find a “Brandt” earlier, we need to search further, since there might be more people named “Brandt”
    • and the query is expected to return them all
Finding Information in Databases

• How does that work, actually?
  – SELECT * FROM employee WHERE name = 'Brandt'

• Better approach
  – Let’s assume we have sorted the table by name

• We can now apply *binary search*
  – Get element in the middle of the table
  – If the searched element is “smaller”
    • Search the upper half table
  – Else
    • Search the lower half table
Finding Information in Databases

- Binary search
  - Works in $O(\log_2 N)$

- However
  - Sorting the table requires $O(N \cdot \log_2 N)$
  - i.e., complexity for search would also be $O(N \cdot \log_2 N)$

- This pays off only if we sort once and query often
  - Total complexity for $S$ binary searches: $O(S \cdot \log_2 N) + O(N \cdot \log_2 N)$
  - Total complexity for $N$ linear searches: $O(S \cdot N)$
    - i.e., binary search is better if $S > \log_2 N$
    - for 1,000,000 entries: more than 20 searches
Finding Information in Databases

• Binary search
  – Sort & search pays off after $\log_2 N$ searches

• But wait... what if our next query is
  SELECT * FROM employee WHERE salary > 50000
• Now, the table is sorted by name, not salary
  – If we re-sort before every query, it gets even worse than by linear search
Finding Information in Databases

• Naive solution
  – Provide copies of each table sorted by each attribute we may need

• Hey, wait…
  – We’ve always tried to *reduce* redundancy
  – Not to *increase* it…

• More sophisticated solution:
  – Index structures

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>76543</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
</tbody>
</table>
Index Files

• Index files
  – Provide a compromise between re-sorting
  – and copying the table

• Idea
  – Provide a sorted file of a single attribute only
    • Allows linear search
  – Index file contains pointers to actual file
    • Which may or may not be sorted
Index Files

• Basic idea
  – Search in index is $O(\log_2 N)$
  – Following link is $O(1)$
  – Each index can remain sorted
  – Create an index for each attribute which you may use in a query

• Trade-off
  – Faster queries
  – Some redundancy
    • But this is handled by the DBMS!
    • i.e., mainly a storage capacity problem, not so much a consistency problem

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
</tbody>
</table>
Index Files and Joins

• Understanding the need for an index file
  – Analyzing possible queries
• First use case: search attributes
  – quite straightforward
• Second use case: joins
• Suppose we want to query for the building of an instructor by name
  – name on instructor is straightforward for an index candidate
  – Query processing:
    • find instructor by name
    • read dept_name
    • look up dept_name in department

hence, we need an index on dept_name in department!
Index Files – Basic Concepts

• Indexing mechanisms used to speed up access to desired data
  – e.g., searching by a specific attribute
  – but also: joins!

• Search Key - attribute to set of attributes used to look up records in a file
  – An index file consists of records (called index entries) of the form
  \[
  \begin{array}{|c|c|}
  \hline
  \text{search-key} & \text{pointer} \\
  \hline
  \end{array}
  \]

• Two basic kinds of indices:
  – Ordered indices: search keys are stored in sorted order
  – Hash indices: search keys are distributed uniformly across “buckets” using a “hash function”
Metrics for Evaluating Index Structures

• Access types supported efficiently
  – records with a specified value in the attribute
  – or records with an attribute value falling in a specified *range* of values

• Access time

• Insertion time
  – Note: index needs to be updated as well

• Deletion time
  – Note: may require deletion from index

• Storage space overhead
Ordered Indices

• In an ordered index, index entries are stored sorted on the search key value
  – allows $O(\log_2 N)$ search

• Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  – Also called clustering index
  – Search key: usually (but not necessarily) the primary key

• Secondary index: an index whose search key specifies an order different from the sequential order of the file
  – Also called non-clustering index
Dense vs. Sparse Index Files

- Dense index: index record appears for every search-key value
  - e.g., index on *ID* attribute of *instructor* relation

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>58583</td>
<td>Califfieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
</tbody>
</table>
Dense vs. Sparse Index Files

- Dense index: index record appears for every search-key value
  - e.g., index on department attribute of instructor relation
Dense vs. Sparse Index Files

- **Sparse Index**: contains index records for only some values
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value $K$ we:
  - Find index record with largest search-key value $< K$
  - Search file sequentially starting at that record
Dense vs. Sparse Index Files

• Dense index
  – Guaranteed search time of $O(\log_2 N)$
  – Requires $O(N)$ storage space

• Sparse index (storing every k-th value)
  – Search time $O(\log_2 N + \log_2 k)$
  – Requires $O(N/k)$ storage time

• Comparison
  – Dense index is faster
  – Sparse index takes less space
Secondary Index

• Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition
  – Example 1: In the instructor relation stored sequentially by ID, we may want to find all instructors in a particular department
  – Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values

• We can have a secondary index with an index record for each search-key value
Secondary Index

- Primary index: index on the attribute by which a file is ordered
- Secondary index: index on any other attribute
  - Index record points to a bucket that contains pointers to all the actual records with that particular search-key value
  - Secondary indices have to be dense — why?

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
</tbody>
</table>
Multi-Level Indices

• Computer storage:
  – RAM: fast, but limited
  – Disk: slow, but large

• Fast access
  – Keep primary index in memory, actual data on disk

• What if the primary index does not fit in memory?
  – Treat primary index kept on disk as a sequential file
  – Construct a sparse index on it, keep that index in memory

• Outer vs. inner index
  – outer index – a sparse index of primary index
  – inner index – the primary index file
Insertion into Index

• Single-level index insertion
  – Perform a lookup using the search-key value appearing in the record to be inserted
  – Dense indices – if the search-key value does not appear in the index, insert it
  – Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created
    • If a new block is created, the first search-key value appearing in the new block is inserted into the index

• Multilevel insertion: algorithms are simple extensions of the single-level algorithms

Costly!
Deletion from Index

• If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also

• Single-level index entry deletion:
  – Dense indices – deletion of search-key is similar to file record deletion
  – Sparse indices
    • if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order)
    • If the next search-key value already has an index entry, the entry is deleted instead of being replaced

• Multilevel deletion: algorithms are simple extensions of the single-level algorithms
Summary Sequential Indices

- Access time: $O(\log_2 N)$
- Insertion time: $O(N)$
  - worst case: insertion at the top, all other entries need to be moved down
- Deletion time: $O(N)$
  - worst case: deletion from the top, all other entries need to be moved up
B⁺-Tree Index Files

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created
  - periodic reorganization of entire file is required
- Advantage of B⁺-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions
  - reorganization of entire file is not required to maintain performance
- (Minor) disadvantage of B⁺-trees:
  - extra insertion and deletion overhead, space overhead
- Advantages of B⁺-trees outweigh disadvantages
- B⁺-trees are used extensively
B⁺-Trees

• A B⁺-tree is a rooted tree satisfying the following properties:
  – All paths from root to leaf are of the same length
  – Each node that is not a root or a leaf has between \( \lceil n/2 \rceil \) and \( n \) children
  – A leaf node has between \( \lceil (n-1)/2 \rceil \) and \( n-1 \) values

• Special cases:
  – If the root is not a leaf, it has at least 2 children.
  – If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and \( (n-1) \) values.
B⁺-Trees: Example
B⁺-Trees: Example

- Example: n=4
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil = 2$ and $n=4$ children
  - A leaf node has between $\lceil (n-1)/2 \rceil = 2$ and $n-1=3$ values
  - Root has at least 2 children
B⁺-Tree Node Structure

• Typical node

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$K_1$</td>
<td>$P_2$</td>
<td>...</td>
<td>$P_{n-1}$</td>
<td>$K_{n-1}$</td>
<td>$P_n$</td>
</tr>
</tbody>
</table>

• $K_i$ are the search-key values

• $P_i$ are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

• The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

– for the moment: assuming there are no duplicate keys, but extension to handling duplicate keys is easily possible
Leaf Nodes in $B^+$-Trees

- For $i = 1, 2, \ldots, n-1$, pointer $P_i$ points to a file record with search-key value $K_i$.
- If $L_i, L_j$ are leaf nodes and $i < j$, $L_i$’s search-key values are less than or equal to $L_j$’s search-key values.
- $P_n$ points to next leaf node in search-key order.

<table>
<thead>
<tr>
<th>leaf node</th>
<th>Pointer to next leaf node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brandt</td>
<td>Califiere</td>
</tr>
<tr>
<td>Crick</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>80000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>58583</td>
<td>Califiere</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
</tbody>
</table>
Inner Nodes in $B^+$-Trees

- Properties of an inner node with $m$ entries:
  - All the search-keys in the subtree to which $P_1$ points are less than $K_1$
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which $P_i$ points have values greater than or equal to $K_{i-1}$ and less than $K_i$
  - All the search-keys in the subtree to which $P_n$ points have values greater than or equal to $K_{n-1}$

| $P_1$ | $K_1$ | $P_2$ | $\ldots$ | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
Observations about B⁺-Trees

• Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close

• The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices

• The B⁺-tree contains a relatively small number of levels
  – Level below root has at least $2^{\lceil n/2 \rceil}$ values
  – Next level has at least $2^{\lceil n/2 \rceil} \cdot 2^{\lceil n/2 \rceil}$ values
    • .. etc.
  – If there are $K$ search-key values in the file, the tree height is no more than $\lceil \log_{n/2}(K) \rceil$
    • thus searches can be conducted efficiently

• Insertions and deletions to the main file can be handled efficiently (as we shall see)
Querying $B^+$-Trees

- Given a search value $V$ (e.g., “Einstein”)
  - In non-leaf nodes: follow non-null pointers $P_i$ where $V<K_i$, so that $i$ maximal
  - In leaf nodes: if there is a value $K_i=V$, follow $P_i$
    - else: record does not exist
Querying B$^+$-Trees

- If there are $K$ search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - i.e., this is the number of leaf nodes to inspect
  - supposing a disk-based index: the number of nodes to be retrieved
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and $n$ is typically around 100 (40 bytes per index entry)
- With 1 million search key values and $n = 100$
  - at most $\log_{50}(1,000,000) = 4$ nodes are accessed in a lookup

Disk I/O is the crucial factor here
Updates on B⁺-Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already present in the leaf node
  - add record to the file
  - if necessary, add a pointer to the bucket
- If the search-key value is not present, then
  - add the record to the main file (and create a bucket if necessary)
  - If there is room in the leaf node
    - insert (key-value, pointer) pair in the leaf node
  - else
    - split the node (along with the new (key-value, pointer) entry)

hopefully the frequent case
Updates on B⁺-Trees: Insertion

- Splitting a leaf node:
  - take the \( n \) (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first \( \lceil n/2 \rceil \) in the original node, and the rest in a new node \( p \)
  - let \( k \) be the least key value in \( p \). Insert \( (k,p) \) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up
- Splitting of nodes proceeds upwards till a node that is not full is found
  - In the worst case (i.e., root is full) the root node may be split increasing the height of the tree by 1

Result of splitting node containing Brandt, Califieri, Crick on inserting Adams

Next step: insert entry with (Califieri,pointer-to-new-node) into parent
Updates on $B^+$-Trees: Insertion

- Inserting “Adams”
Updates on B⁺-Trees: Insertion

- Inserting “Lamport”
Updates on B⁺-Trees: Deletion

• Find the record to be deleted, and remove it from the main file and from the bucket (if present)
• Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
• If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then *merge siblings*
• Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then *redistribute pointers*
Updates on B⁺-Trees: Deletion

- Deleting “Srinivasan”

Diagram showing the process of deleting a node labeled “Srinivasan” from a B⁺-tree, updating the tree structure accordingly.
Indices on Multiple Attributes

- Use multiple indices for certain types of queries
- Example:
  
  ```
  select ID  
  from instructor  
  where dept_name = "Finance" and salary = 80000
  ```
- Possible strategies for processing query using indices on single attributes:
  1. Use index on `dept_name` to find instructors with department name Finance; test `salary = 80000`
  2. Use index on `salary` to find instructors with a salary of $80000; test `dept_name = "Finance"`
  3. Use both indices, take intersection of sets of pointers obtained
Indices on Multiple Attributes

• Composite search keys are search keys containing more than one attribute
  – e.g. (dept_name, salary)

• Lexicographic ordering: \((a_1, a_2) < (b_1, b_2)\) if either
  – \(a_1 < b_1\), or
  – \(a_1 = b_1\) and \(a_2 < b_2\)

• Use this ordering to create an index (sequential or B*-tree)
Indices on Multiple Attributes

- Suppose we have an index on (dept_name, salary)
- With the **where** clause
  
  ```
  where dept_name = “Finance” and salary = 80000
  ```
  the index on (dept_name, salary) can be used to fetch only records that satisfy both conditions
- Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions
Indices on Multiple Attributes

• Note:
  – Ordering is sensitive to order of attributes
  – i.e., (salary,dept_name) would lead to a different ordering!

• With (dept_name,salary), we can efficiently retrieve
  \[\text{dept}\_\text{name} = \text{"Finance" and salary} \geq 80000\]

• But not
  \[\text{dept}\_\text{name} > \text{"Finance" and salary} = 80000\]

• Ordering of index is by dept_name first, then salary
Multi-Attribute Indices vs. Multiple Indices

• Multi-Attribute are faster than multiple indices
  – Make sure you only retrieve the records you are interested in
  – Avoid unnecessary lookups, comparisons, and/or intersections

• On the other hand
  – Storing an index for all combinations of attributes would be costly
    • 10 attributes, all combinations of only 2 attributes → 100 indices!
  – Think: storage capacity
  – Think: cost of insert/update/delete operations

• Typical considerations
  – Heavily used attribute combinations
  – Expected runtime disadvantage of individual indices
Indexing vs. Hashing

- Index structures:
  - Look up value
  - Retrieve storage location (e.g., row number in table)

- Hashing:
  - Compute storage location directly from the value using a hash function
Static Hashing

• Bucket: unit of storage containing one or more records
  – Typically: a disk block
• Hash function \( h \): maps a search key to the block where the record is located
  – \( h : K \rightarrow B \)
  – Records with different search-key values may be mapped to the same bucket
    → bucket has to be searched sequentially to eventually locate a record
    → bucket overflow occurs when a bucket is full
Example for a Hash Function

- There are 10 buckets
- The hash function maps a department name to numbers between 0-9
- e.g., $h(\text{Music}) = 1$  $h(\text{History}) = 2$
  $h(\text{Physics}) = 3$  $h(\text{Elec. Eng.}) = 3$
Hash Functions

• A hash function should be
  – *uniform*, i.e., each bucket is assigned the same number of search-key values
  – *random*, i.e., the size of buckets should be independent of the actual distribution of search-key values
    • e.g., language is not uniformly distributed

• Worst case: all search-key values map to the same bucket
  – access time proportional to the number of search-key values in the file

• Typical hash functions perform computation on the internal binary representation of the search-key
  – e.g., for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned
Bucket Overflow

- **Overflow chaining** (also called **closed hashing**)
  - the overflow buckets of a given bucket are chained together in a linked list
  - slows search for actual record
  - cannot be entirely avoided, but reduced by good choice of hash function

![Diagram of bucket overflow chaining](image-url)
Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation
  - A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure
Drawbacks of Static Hashing

- In static hashing, function $h$ maps search-key values to a fixed set of $B$ of bucket addresses
  - But databases may grow or shrink over time
- Growing database
  - Performance degrades due to many overflow buckets
- Shrinking database
  - Space is wasted by underfull buckets
- Possible solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution
  - Allow the number of buckets to be modified dynamically
  - Aka dynamic hashing
Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range
  - Typically $b$-bit integers, e.g., $b = 32$.
- At any time use only a prefix of the hash function to index into a table of bucket addresses
  - Let the length of the prefix be $i$ bits, $0 \leq i \leq 32$.
  - Bucket address table size = $2^i$. Initially $i = 0$
- Value of $i$ grows and shrinks as the size of the database grows and shrinks
- Multiple entries in the bucket address table may point to a bucket (why?)
  - Thus, actual number of buckets is $< 2^i$
  - Number of buckets also changes dynamically by merging and splitting buckets
Extendable Hash Structure

- Example:
  - more hash values with prefix “1” than with prefix “0”
Extendable Hashing

• Each bucket $j$ stores a value $i_j$
• All the entries that point to the same bucket have the same values on the first $i_j$ bits
• To locate the bucket containing search-key $K_j$:
  1. Compute $h(K_j) = X$
  2. Use the first $i$ bits of $X$ as a displacement into bucket address table, and follow the pointer to appropriate bucket
• Insertion and deletion may cause splitting/merging of buckets
• Overflow buckets may still be needed for key collisions
Extendable Hashing – Example

- Bucket size: 2

<table>
<thead>
<tr>
<th>dept_name</th>
<th>h(dept_name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>0010 1101 1111 1011 0010 1100 0011 0000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>1111 0001 0010 0100 1001 0011 0110 1101</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>0100 0011 1010 1100 1100 0110 1101 1111</td>
</tr>
<tr>
<td>Finance</td>
<td>1010 0011 1010 0000 1100 0110 1001 1111</td>
</tr>
<tr>
<td>History</td>
<td>1100 0111 1110 1101 1011 1111 0011 1010</td>
</tr>
<tr>
<td>Music</td>
<td>0011 0101 1010 0110 1100 1001 1110 1011</td>
</tr>
<tr>
<td>Physics</td>
<td>1001 1000 0011 1111 1001 1100 0000 0001</td>
</tr>
</tbody>
</table>

Bucket 0
Extendable Hashing – Example

- After insertion of Mozart, Srinivisan, Wu

Prefix length 1

Bucket 0

<table>
<thead>
<tr>
<th>15151</th>
<th>Mozart</th>
<th>Music</th>
<th>40000</th>
</tr>
</thead>
</table>

Bucket 1

<table>
<thead>
<tr>
<th>10101</th>
<th>Srinivisan</th>
<th>Comp.Sci</th>
<th>90000</th>
</tr>
</thead>
<tbody>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
</tbody>
</table>
Extendable Hashing – Example

- After insertion of Einstein

Pointers to same bucket

<table>
<thead>
<tr>
<th>Bucket 0</th>
<th>15151</th>
<th>Mozart</th>
<th>Music</th>
<th>40000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bucket 1</td>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>Bucket 2</td>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp.Sci</td>
<td>90000</td>
</tr>
</tbody>
</table>
Extendable Hashing – Example

- After insertion of Gold, El Said

<table>
<thead>
<tr>
<th>Bucket 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15151</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>22222</td>
</tr>
<tr>
<td>33456</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>12121</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
</tr>
<tr>
<td>32343</td>
</tr>
</tbody>
</table>
## Extendable Hashing – Example

- **After inserting Feinman**

  **Bucket 0**
<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Department</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
</tbody>
</table>

  **Bucket 1**
<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Department</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
</tbody>
</table>

  **Bucket 1a**
<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Department</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>47035</td>
<td>Feinman</td>
<td>Physics</td>
<td>92000</td>
</tr>
</tbody>
</table>

  **Bucket 2**
<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Department</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
</tbody>
</table>

  **Bucket 3**
<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Department</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivisan</td>
<td>Comp.Sci</td>
<td>90000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
</tbody>
</table>
Extendable Hashing

• Benefits
  – Hash performance does not degrade with growth of file
  – Minimal space overhead
• Disadvantages
  – Extra level of indirection to find desired record
  – Bucket address table may itself become very big (larger than memory)
    • Cannot allocate very large contiguous areas on disk either
    • Solution: B+-tree structure to locate desired record in bucket address table
  – Changing size of bucket address table is an expensive operation
Comparison of Indexing and Hashing

• Expected type of queries:
  – Hashing is generally better at retrieving records having a specified value of the key.
  – If range queries are common, ordered indices are to be preferred
• Cost of periodic re-organization
• Relative frequency of insertions and deletions
• Average vs. worst case access time
• Which index type is supported by the DBMS at hand?
Bitmap Indices

• Special type of index designed for efficient querying on multiple keys
• Records in a relation are assumed to be numbered sequentially from, say, 0
  – Given a number \( n \) it must be easy to retrieve record \( n \)
• Applicable on attributes that take on a relatively small number of distinct values
  – e.g. gender, country, state, …
  – e.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
• A bitmap is simply an array of bits
• CPUs can process them very efficiently (i.e., 32 or 64 bits at once)
Bitmap Indices

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

<table>
<thead>
<tr>
<th>record number</th>
<th>ID</th>
<th>gender</th>
<th>income_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>76766</td>
<td>m</td>
<td>L1</td>
</tr>
<tr>
<td>1</td>
<td>22222</td>
<td>f</td>
<td>L2</td>
</tr>
<tr>
<td>2</td>
<td>12121</td>
<td>f</td>
<td>L1</td>
</tr>
<tr>
<td>3</td>
<td>15151</td>
<td>m</td>
<td>L4</td>
</tr>
<tr>
<td>4</td>
<td>58583</td>
<td>f</td>
<td>L3</td>
</tr>
</tbody>
</table>

Bitmaps for gender

- m: 10010
- f: 01101

Bitmaps for income_level

- L1: 10100
- L2: 01000
- L3: 00001
- L4: 00010
- L5: 00000
Bitmap Indices

• Bitmap indices are useful for queries on multiple attributes
  – not particularly useful for single attribute queries
• Queries are answered using bitmap operations
  – Intersection (and)
  – Union (or)
  – Negation (not)
• Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  – Males with income level L1: 10010 AND 10100 = 10000
  – People with income level L3 to L5: 00001 OR 00010 OR 00000 = 00011
  – Females with income above L1: 01101 AND (NOT 10100) = 01001
• Can then retrieve required tuples
  – Counting number of matching tuples is even faster!
Selected Other Index Types

- Tries (also known as Prefix Trees)

![Tries Diagram]

- Select examples from Google Search:
  - google is
  - google is evil
  - google is god
  - google is your friend
  - google is skynet
  - google is acting weird
  - google is down
  - google is awesome
  - google is taking over the world
  - google is watching you
  - google is cia
Selected Other Index Types

- R-Trees and kd trees
Summary

• Index structures help making queries efficient
  – Practically, speedup by many orders of magnitude
• Trading off storage against computation time
• We’ve got to know different flavors
  – Table index
  – $B^+$-Tree
  – Hash tables
  – Bitmap indices
• Choice of an index structure
  – Desired queries (single/multi attribute? range or value? counting?)
  – Frequency of updates
  – Real time requirements
Questions?