Part IX

B-Trees
Memory Hierarchy

- fast, but expensive and small, memory close to CPU
- larger, slower memory at the periphery
- Try to **hide latency** by using the fast memory as a **cache**.
“Slow” memory typically means **high latency**.

**Example:** Samsung HD642JJ Hard Drive (640 GB, SATA 3)
- rotational speed: 7200 rpm
- sequential read bandwidth: \( \approx 106 \text{ MB/s} \) (\( \uparrow \text{hdparm} \ -t \))
- random access time: 15.2 ms (measured)

**Time it takes to read 1,024 random 4 kB blocks?**
Ways to Improve I/O Performance

The latency penalty is hard to avoid.

However:

- Throughput can be increased rather easily by exploiting parallelism.
- Idea: Use multiple disks and access them in parallel.

TPC-C: An industry benchmark for OLTP

The current number one system (Oracle 11g RAC on SPARC) uses

- 11,040 flash drives (24 GB each) and 720 hard drives (!) (plus drives for OS, etc.),
- connected with 8 Gbit Fibre Channel,
- yielding 30 tpmC (∼60 M transactions per minute).
Consequences of the Bandwidth ↔ Latency Gap

To combat the latency problem:

1. Databases access and organize the disk with a page granularity.
   - Read larger chunks to amortize high latency.
   - Page size: at least 4 kB, better more; up to \( \approx 64 \text{ kB} \).

2. Use sequential access and/or aggressive prefetching (read-ahead).
   - But must read many pages ahead to actually avoid penalty.
To answer this query, we could

1. **sort** the table on disk (in ZIPCODE order).
2. To answer queries, then use **binary search** to find first qualifying tuple, and **scan** as long as ZIPCODE < 8999.

\[ k^* \] denotes the full data record with search key \( k \).
Ordered Files and Binary Search

<table>
<thead>
<tr>
<th>Page</th>
<th>Data Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4104*</td>
</tr>
<tr>
<td>1</td>
<td>4123*</td>
</tr>
<tr>
<td>2</td>
<td>4222*</td>
</tr>
<tr>
<td>3</td>
<td>4450*</td>
</tr>
<tr>
<td>4</td>
<td>5012*</td>
</tr>
<tr>
<td>5</td>
<td>6330*</td>
</tr>
<tr>
<td>6</td>
<td>6423*</td>
</tr>
<tr>
<td>7</td>
<td>8050*</td>
</tr>
<tr>
<td>8</td>
<td>8105*</td>
</tr>
<tr>
<td>9</td>
<td>8180*</td>
</tr>
<tr>
<td>10</td>
<td>8245*</td>
</tr>
<tr>
<td>11</td>
<td>8280*</td>
</tr>
<tr>
<td>12</td>
<td>8406*</td>
</tr>
</tbody>
</table>

Need to read only $\log_2(\#\; tuples)$ to find the first match.

Need to read about as many **pages** for this.
(The whole point of binary search is that we make far, unpredictable jumps. This largely defeats prefetching.)
Observations:

- Make rather **far jumps initially**.
  - For each step read **full page**, but inspect only **one record**.
- Last $O(\log_2 \text{pagesize})$ steps stay **within one page**.
  - I/O cost is used much more efficiently here.
**Idea:** “Cache” those records that might be needed for the first phase.

→ If we can keep the cache **in memory**, we can find **any** record with just a **single I/O**.

⚠️ **Is this assumption reasonable?**
What if my data set is really large?

- “Cache” will span many pages, too.
  (In practice, we’ll organize the cache just like any other database object.)
- Thus: “cache the cache” → hierarchical “cache”

B-trees are essentially such a “hierarchical cache.”
B-Trees

- All nodes are the size of a page
  - hundreds of entries per page
  - large fanout, low depth
- Search effort: $\log_{\text{fanout}}(\# \text{ tuples})$

B-Trees

Each B-tree node contains

- A set of **index entries**, which include
  - the value of a **search key** (e.g., 4711) and
  - “**associated information**” (indicated by *)
    (either a full data tuple or a reference to a tuple).
- A set of **child pointers**, pointing to a child page of the B-tree.

Each tree node (except the root) contains **at least** $d$ and **at most** $2d$ index entries ("minimum 50 % full"; on previous slide: $d = 2$).

→ We call $d$ the **order** of the B-tree.
→ In practice, $d$ is **large** (few hundreds).

B-trees are **balanced** at all times.
Searching a B-Tree

Function: tree_search \( (k, node) \)

1. if matching \( *_i \) found on \( node \) then
   - return \( *_i \);
2. if \( node \) is a leaf node then
   - return not found;
3. switch \( k \) do
   - case \( k < k_0 \) do
     - return tree_search \( (k, p_0) \);
   - case \( k_i < k < k_{i+1} \) do
     - return tree_search \( (k, p_i) \);
   - case \( k_{2d} < k \) do
     - return tree_search \( (k, p_{2d}) \);

- Invoke with \( node = \) root node.
- Note that B-trees are an ordered index structure.
  - Support equality and range predicates
Goal: Keep B-tree balanced at all times.\textsuperscript{14}

Why is this desirable?

Thus: Define routines for \texttt{insertion} and \texttt{deletion} that leave the B-tree properties intact.

\textsuperscript{14}/i.e., every root-to-leaf path must have the same length.
Inserting into a B-Tree

Sketch of the insertion procedure for entry $k*$:

1. **Find leaf page** $n$ where we would expect the entry for $k$.

2. If $n$ has **enough space** to hold the new entry (*i.e.*, at most $2d - 1$ entries in $n$), **simply insert** $k*$ into $n$.

3. Otherwise node $n$ must be **split** into $n$ and $n'$ and a new **separator** has to be inserted into the parent of $n$.

Splitting happens recursively and may eventually lead to a split of the root node (increasing the tree height).

→ B-trees grow at the root, not at the leaves!
Insert new entry with key 4222.

→ Enough space in node 3, simply insert.
Insert key **6330**.

→ Must **split** node 4.

→ **Middle entry** goes into node 1.
After 8180, 8245, 6435 insert key **4104**.

→ Must **split** node 3.

→ Node 1 overflows → split it

→ **New separator** goes into root
Splitting starts at the leaf level and continues upward as long as index nodes are fully occupied.

Eventually, this can lead to a split of the root node:
- Split like any other inner node.
- Use the separator to create a new root.

The root node is the only node that may have an occupancy of less than 50%.

This is the only situation where the tree height increases.

How often do you expect a root split to happen?
A B-tree maintains **key values** together with "**associated information**".

The "associated information" *can be*

**Full Data Tuples**

The B-tree becomes the mechanism to organize the table data

→ The table is **physically ordered** according to the key attribute.
→ We call this a **clustered index** or an **index-organized table**.
→ There can be at most one clustered index per table.

**Pointers to Actual Tuples**

These pointers are also called **record identifiers** or **RIDs**.

→ Most systems use ⟨*pageno*, *pos. within page*⟩ to encode RIDs.
→ Such indexes are also called **secondary indexes**.
→ There can be arbitrarily many secondary indexes.

Many systems (*e.g.*, DB2) only support the latter index type.
Key to the efficiency of B-trees is their high fanout.

high fanout → low tree depth → fast root-to-leaf navigation

This gives incentive to maximize fanout:

→ Do not store * in inner nodes
  (Rather use that space to increase \(d\) / store more keys.)
→ Inner nodes are then used for root-to-leaf navigation only.
→ For every data tuple, there is on leaf-level index entry.
→ The resulting index structure is then called \(B^+\)-tree.

**Real systems** today always use \(B^+\)-trees.

→ When database people say “B-tree,” they typically mean “\(B^+\)-tree.”
B⁺-trees

- Inner nodes do **not** store tuples or RIDs
  - only used to navigate to leaves
  - higher fanout, lower depth
- Only leaves contain (references to) tuple data (indicated here with *)
Searching a $\mathbb{B}^+$-tree

1. **Function**: `search(k)`
   2. `return tree_search(k, root);`

   All searches now navigate to a leaf node.

   → Makes search effort also more predictable.

2. **Function**: `tree_search(k, node)`
   3. `if node is a leaf then`
      4. `return node;`
   5. `switch k do`
      6. `case k < k_0 do`
      7. `return tree_search(k, p_0);`
      8. `case k_i < k < k_{i+1} do`
      9. `return tree_search(k, p_i);`
     10. `case k_{2d} < k do`
     11. `return tree_search(k, p_{2d});`

Function `search(k)` returns a pointer to the leaf node that contains potential hits for search key $k$. 
Insert new entry with key 4222.

→ Enough space in node 3, simply insert. (Same as in B-tree)
**B⁺-tree Insert: Examples (Insert with Leaf Split)**

Insert key **6330**.

→ **Must split** node 4.

→ **New separator** goes into node 1.

But **keep** entry in node 4!
After 5219, 5476, insert key 4104.

→ Must **split** leaf node 3.
→ Inner node 1 overflows → split it
→ **New separator** goes into root

Splitting the **inner node** works analogously to B-tree.
Function:  tree_insert$(k, rid, node)$

if $node$ is a leaf then

return leaf_insert$(k, rid, node)$;

else

switch $k$ do

  case $k \leq k_0$ do

  $\langle sep, ptr \rangle \leftarrow$ tree_insert$(k, rid, p_0)$;

  case $k_i < k \leq k_{i+1}$ do

  $\langle sep, ptr \rangle \leftarrow$ tree_insert$(k, rid, p_i)$;

  case $k_{2d} < k$ do

  $\langle sep, ptr \rangle \leftarrow$ tree_insert$(k, rid, p_{2d})$;

  if $sep$ is null then

  return $\langle null, null \rangle$;

  else

  return split$(sep, ptr, node)$;

see tree_search()
Function: leaf_insert \((k, rid, node)\)

1. If another entry fits into \(node\) then
   2. Insert \(\langle k, rid \rangle\) into \(node\);
   3. Return \(\langle \text{null}, \text{null} \rangle\);

else

4. Allocate new leaf page \(p\);
5. Take \(\{\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\} := \) entries from \(node \cup \{\langle k, ptr \rangle\}\)
6. Leave entries \(\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle\) in \(node\);
7. Move entries \(\langle k_{d+2}^+, p_{d+2}^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\) to \(p\);
8. Return \(\langle k_{d+1}^+, p \rangle\);

Function: split \((k, ptr, node)\)

1. If another entry fits into \(node\) then
   2. Insert \(\langle k, ptr \rangle\) into \(node\);
   3. Return \(\langle \text{null}, \text{null} \rangle\);

else

4. Allocate new leaf page \(p\);
5. Take \(\{\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\} := \) entries from \(node \cup \{\langle k, ptr \rangle\}\)
6. Leave entries \(\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle\) in \(node\);
7. Move entries \(\langle k_{d+2}^+, p_{d+2}^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\) to \(p\);
8. Set \(p_0 \leftarrow p_1^+\) in \(node\);
9. Return \(\langle k_{d+1}^+, p \rangle\);
Function:  

\[
\langle \text{key}, \text{ptr} \rangle \leftarrow \text{tree_insert}(k, \text{rid}, \text{root});
\]

if key is not null then

allocate new root page \( \text{r} \);

\begin{align*}
\text{populate } n \text{ with} \\
p_0 & \leftarrow \text{root}; \\
k_1 & \leftarrow \text{key}; \\
p_1 & \leftarrow \text{ptr}; \\
\text{root} & \leftarrow \text{r} ;
\end{align*}

\begin{itemize}
\item \text{insert} \((k, \text{rid})\) is called from outside.
\item Note how leaf node entries point to \text{RID}s, while inner nodes contain pointers to other \(B^+\)-tree nodes.
\end{itemize}
Example: Webserver access log (people.inf.ethz.ch)
- table cardinality: 11 million tuples (710K data pages)
- distinct IP addresses: 181,628 (stored as CHAR (15))
- database: IBM DB2 9.7

$B^+$-tree on IP addresses:
- 25,151 index pages total:
  - 1 root node
  - 110 second-level nodes; average fanout: 230
  - 25,040 leaf-level nodes: 1–77 keys per node
If a node is sufficiently full (i.e., contains at least $d + 1$ entries, we may simply remove the entry from the node. Note: Afterward, inner nodes may contain keys that no longer exist in the database. This is perfectly legal.

- **Merge** nodes in case of an underflow ("undo a split"): “Pull” separator into merged node.
Deletion

It’s not quite that easy...

- Merging only works if **two** neighboring nodes were 50 % full.
- Otherwise, we have to **re-distribute**:
  - “rotate” entry through parent
- Redistribution is **complex** and **expensive**.
  → Real systems usually do not implement deletion “by the book.”
Actual systems often avoid the cost of merging and/or redistribution, but relax the minimum occupancy rule.

*E.g.*, **IBM DB2 UDB**:

- The `MINPCTUSED` parameter controls when the system should try a leaf node merge ("on-line index reorg").
- Inner nodes are never merged (→ need to do full table reorg for that).

To improve **concurrency**, systems sometimes only **mark** index entries as deleted and physically remove them later (*e.g.*, IBM DB2 UDB “type-2 indexes”)

→ Resulting index entries are also called **ghost records**.
A typical situation (for a secondary $\mathbf{B^+}$-tree) looks like this:

What are the implications when we want to execute

```
SELECT * FROM CUSTOMERS ORDER BY ZIPCODE
```
Composite Keys

$B^+$-trees can (in theory\(^\text{15}\)) be used to index everything with a defined total order, e.g.:

- integers, strings, dates, . . . , and
- concatenations thereof (based on lexicographical order).

*E.g.* in most SQL dialects:

```
CREATE INDEX ON TABLE CUSTOMERS (LASTNAME, FIRSTNAME);
```

A useful application are, *e.g.*, partitioned $B$-trees:

- Leading index attributes effectively partition the resulting $B^+$-tree.


\(^{15}\)Some implementations won’t allow you to index, *e.g.*, large character fields.
CREATE INDEX ON TABLE STUDENTS (SEMESTER, ZIPCODE);

What types of queries could this index support?