Relational Database System Implementation

CS122 – Lecture 15

Winter Term, 2017-2018

Index Optimizations

- So far, only discussed implementing relational algebra operations to directly access heap files
- Indexes present an alternate *access path* for finding specific records
- Use indexes to create optimized implementations of relational algebra operations
- Apply index-based accesses in query plans where it makes sense to do so
  - Optimizer needs strategies for how to use indexes
  - Also needs accurate cost-estimates for index accesses

B$^+$-Tree Index Optimizations

- Will focus primarily on B$^+$-tree indexes
  - Virtually all databases have B$^+$-tree indexes
  - Other kinds of indexes are far less common
  - (Not hard to figure out the details yourself, if curious…)
- Virtually all database indexes are *secondary indexes*
  - Order of index-entries does not correspond to order of records in data file (which is usually a heap file)
  - *Primary indexes* are in same search-key order as the file they are built against
- **Looking up records referenced by the secondary index will likely incur many additional disk-seeks**

Index Scans

- Previously discussed file scans
  - Scan through entire table file, evaluating predicate against every record
  - If predicate involves equality against a primary key, can stop when we find the record
- If a suitable index exists on columns referenced in the selection predicate, can perform an *index scan* instead
  - Evaluate some portion of the predicate against the index, to identify which rows in the table to retrieve
  - Index contains *pointers* to the records to retrieve
  - If a suitable index is not available, must use a file scan

Index Scans (2)

- Can use indexes for different kinds of predicates
- B$^+$-tree indexes:
  - Equality-based lookups
    - SELECT * FROM employees WHERE emp_id = 352103;
  - Comparisons (a.k.a. *range queries*)
    - SELECT * FROM employees WHERE annual_salary >= 85000;
- Hash indexes:
  - Equality-based lookups only
- Planner/optimizer must understand what kinds of predicates can be optimized with different indexes

Index Scan: Equality on Key

- Index scan; equality on candidate-key attribute:
  - Know that we will retrieve at most one record from table
- Procedure (and associated costs, worst case):
  - Starting with root node in index, navigate the B^+-tree to find the entry for the record
    - One disk seek and one block-read, for each level in the tree
  - Finally, use index's record-pointer to retrieve the record
    - One more disk seek, and one more block-read
- Worst-case estimate: $h_i + 1$ seeks, $h_i + 1$ block-reads
  - h_i denotes the height of the index

Index Scan: Equality on Key (2)

- Optimizers can often assume much faster index access
  - For a given B^+-tree, not unusual for non-leaf nodes to comprise less than 1-2% of the tree
  - Root, and many non-leaf nodes, will likely be in memory
  - Optimizers can probably assume that only the leaf nodes will need to be loaded from disk
- Previous estimate: $h_i + 1$ seeks, $h_i + 1$ block-reads
  - Assumes no index nodes are in memory
- A more optimistic estimate: 2 seeks, 2 block-reads
  - Assumes all non-leaf index nodes are already loaded

Index Scan: Equality on Non-Key

- Can have indexes on non-key columns as well
  - Table may contain many rows with specified value
- Index contains pointers to records in table file
- Worst-case: record-pointers are in random order, and all records are in different blocks
  - Assume *n* is number of records fetched via index
  - Would incur up to *n* disk seeks and *n* block reads, on top of cost of navigating the index (h_i seeks and h_i block-reads)
- Normally it isn't nearly this bad…
  - Required blocks of table file may already be in memory
- If record pointer is used as a uniquifier in the search-key, index entries for a given value will not be in random order!
  - Reading table-records for a given key will incur minimal seeks

Index Scan: Comparisons

- Range scans are also straightforward
  - SELECT * FROM employees WHERE salary > 85000;
- Use index structure to navigate to starting point in sequence of leaf-nodes
- Traverse sequence of leaf-nodes, retrieving records referenced by index entries
- Example: index on employees.salary, in increasing order of salary values
  - Navigate tree to where entries have salary value > 85000
  - Traverse leaf-node entries until entire index is scanned

Index Scan: Comparisons (2)

- Can also perform range-scans with < or ≤ conditions
- Example: index on employees.salary, in increasing order of salary values
  - SELECT * FROM employees WHERE salary < 40000;
- In these cases:
  - Start with smallest search-key value in index
  - Scan through leaf records until ¬(salary < 40000)

- Easily use index to satisfy any comparison (>, ≥, <, ≤)

Index Scan: Comparison Costs

- Can run into same issue as with equality-based index lookup on a non-key column:
  - Index-scan retrieves rows that include record-pointers
  - Index-scan will identify multiple rows
  - Rows are almost certainly <u>not</u> in the same physical order as the logical order specified by the index
- Will likely incur a *large* number of disk seeks:
  - Potentially one seek per record retrieved
    - (usually isn't *this* bad, but still imposes a very heavy cost)
  - Potentially one seek per leaf-block in index, as well
    - (assume we can ignore this, if indexes are well-maintained)

Index Scan: Comparison Costs (2)

- Given:
  - h_i is height of the B$^+$-tree index
  - n rows will match the comparison predicate
  - Index entries for matching rows occupy b leaf-nodes
- Steps and their costs:
  - Navigate to starting-point in sequence of leaf nodes
    - h_i disk seeks and h_i block reads
  - Read through b leaf-nodes
    - b block reads (assume index has leaves in roughly sequential order)
  - Fetch each of n records from table-file
    - Up to a maximum of n disk seeks and n block reads
- Overall worst-case cost: $h_i + n$ seeks, $h_i + n + b$ block-reads

Index Scan: Comparison Costs (3)

- Could apply clever techniques:
  - Read in multiple blocks of index entries that satisfy the selection predicate
  - Sort entries based on record pointers
  - Retrieve the records in that order
- However, results will no longer be in search-key order
  - Not a huge issue, but interferes with Selinger-style optimization techniques
  - Can’t take advantage of records in search-key order further up the plan-tree

Index Scan: Comparison Costs (4)

- Generally, optimizer must choose when to use an index *very* carefully…
- A simple file-scan will read every disk block, but will also incur *far* fewer disk seeks!
  - A disk seek can be as expensive as 10+ sequential block-reads
- Index scan will only save time if a small number of records are being fetched
  - (Use table statistics and costing estimates to guess how many rows a selection predicate might produce.)
- Can still sometimes use indexes for fast range-queries
  - Index entries also contain search-key values…
  - Not every situation requires the entire record to be fetched

Indexes

- Can even satisfy some queries entirely from an index
- Example:
  - SELECT department, AVG(salary) FROM employees GROUP BY department;
  - Two-column index on (department, salary)
- This is often called a *covering index*
- In cases like this, most databases will compute the query entirely against the index
  - Don’t need to incur accesses to the table at all
- (Be aware of this when you design databases, too! ☺)

Indexes and Complex Selections

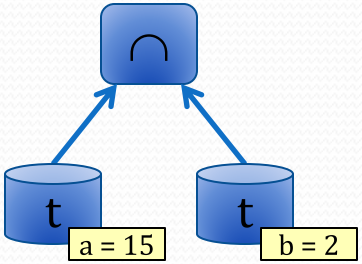
- Often have more complex selections: $\sigma_{P1 \wedge P2 \wedge \ldots}(r)$
  - Conjunctive selection
- Examine individual conditions to determine if an index can be used for any of them
- If a single condition can benefit from an index, e.g. P1:
  - $\sigma_{P1 \wedge P2 \wedge \ldots}(r) = \sigma_{P2 \wedge \ldots}(\sigma_{P1}(r))$
  - Use index to optimize selection on P1, then apply other predicates in memory

Indexes and Complex Selections (2)

- An index's search-key may include multiple attributes
  - If predicate includes multiple comparisons on index-attrs, we can *sometimes* leverage index to speed lookup
- Example: table T with columns A, B, C, D
  - B$^+$-tree index on (A, B, C)
- SELECT * FROM T WHERE A = 5 AND B > 3;
  - Rows satisfying these conditions will be adjacent in the index
- SELECT * FROM T WHERE B = 45 AND C < 12;
  - Can't use index for this predicate ☹
  - Index entries are ordered on A first, then B, and finally C
  - Entries with B = 45 will likely be scattered throughout index

Indexes and Complex Selections (3)

- Complex selections:
  - Conjunctive selection: $\sigma_{P1 \wedge P2 \wedge \ldots}(r)$
  - Disjunctive selection: $\sigma_{P1 \vee P2 \vee \ldots}(r)$
- If we have multiple indexes on input table:
  - Can perform individual selections, then apply set operations to compute complex selection
- Example: $\sigma_{A = 15 \wedge B = 2}(t)$
  - Two different indexes on t: one on A, another on B
  - Perform two index-scans to get record-pointers
  - Use set-intersection on pointers to compute result
    - (For disjunctive selection, use set-union instead)
  - Finally, look up each record using its record-pointer



Join Optimizations

- Joins involve row lookups based on column-values
  - Can frequently leverage indexes to improve performance
- Indexed Nested-Loop Join:

```
for each tuple $t_r$ in $r$:
    using index on $s$, iterate over tuples $t_s$ in $s$
            that satisfy join condition:
        add join($t_r$, $t_s$) to result
```

- Inner loop is effectively performing index-based selection against *s*, based on the join condition
  - Estimate cost of inner-loop lookups based on condition, and on whether attributes are candidate keys or not

Indexed Nested-Loop Join

- Worst case: database can only hold one block of each table in memory
- Indexed Nested-Loop Join:

```
for each tuple $t_r$ in $r$:
    using index on $s$, iterate over tuples $t_s$ in $s$
            that satisfy join condition:
        add join($t_r$, $t_s$) to result
```

- Requires b_r seeks and block-reads for outer loop
- Incurs cost n_r × c for inner loop
  - c = cost of index-based selection; depends on index, the join predicate, etc.

Indexed Nested-Loop Join (2)

- Indexed Nested-Loop Join, worst-case cost:
  - Requires b_r seeks and block-reads for outer loop
  - Incurs cost $n_r \times c$ for inner loop
    - c = cost of index-based selection; depends on index, the join predicate, etc.
- Must be very careful to consider increased seek-costs!
- If an index is available on both sides of join, generally makes sense to put smaller table on outer loop
  - Must perform one index-lookup per row in outer table
  - Large fanout of B^+-tree means index-lookup cost will likely be about same regardless of which is outer table

Hybrid Merge-Join

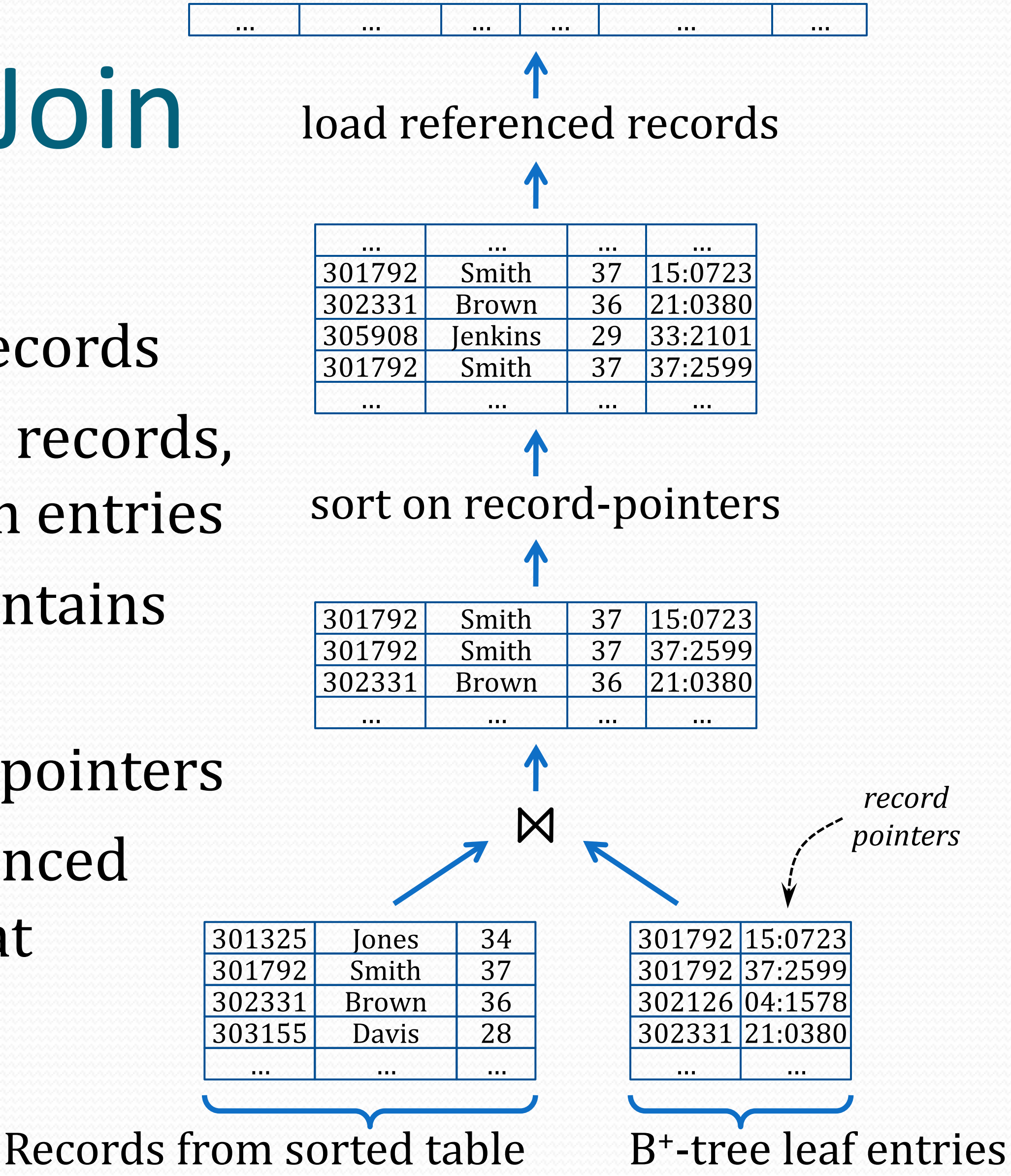
- Sort-merge join requires input relations to be sorted on join-attributes
- Usually will not be the case…
  - Generally store our records in heap files
- But, often have ordered indexes on the join-attributes, on one or both tables involved in the join
- Can use these indexes to perform a *hybrid merge-join*

Hybrid Merge-Join (2)

- Example: one relation is sorted, other is unsorted
  - Have a B$^+$-tree index on join-attrs of unsorted relation
- Procedure:
  - Perform a merge join between records of sorted table and leaf-entries of the B$^+$-tree index on unsorted table
  - Intermediate results contain records from one table, and record-pointers into the other table
  - Sort intermediate results on the record-pointers
    - Minimize disk-seek costs from retrieving referenced records
  - Retrieve and join in the referenced records

Hybrid Merge-Join

- Example:
  - Left table has sorted records
  - B+-tree on right table's records, with record-pointers in entries
  - Merge-joined result contains record pointers
  - Sort results on record-pointers
  - Load and join in referenced records in an order that minimizes disk seeks



…	…	…	…	…	…

load referenced records

…	…	…	…
301792	Smith	37	15:0723
302331	Brown	36	21:0380
305908	Jenkins	29	33:2101
301792	Smith	37	37:2599
…	…	…	…

sort on record-pointers

301792	Smith	37	15:0723
301792	Smith	37	37:2599
302331	Brown	36	21:0380
…	…	…	…

301325	Jones	34
301792	Smith	37
302331	Brown	36
303155	Davis	28
…	…	…

Records from sorted table

301792	15:0723
301792	37:2599
302126	04:1578
302331	21:0380
…	…

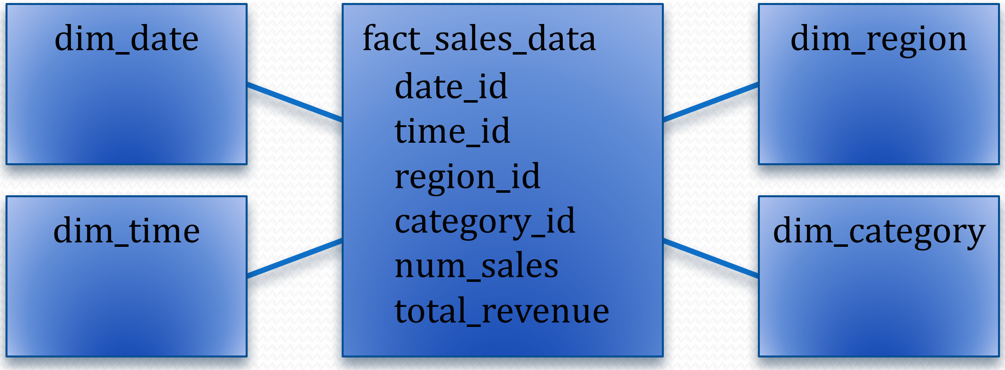
B+-tree leaf entries

Hybrid Merge-Join (4)

- Can easily extend this procedure to merge-join two unsorted relations with appropriate ordered indexes
- Benefits:
  - Index entries are often much smaller than records themselves
  - Sorting the index entries on record-pointers may be *much* more efficient than sorting the actual records
- Drawbacks:
  - Requires multiple sort/load passes to minimize disk seeks
    - Sort on left table's record-pointers to load left table's tuples, then sort on right table's pointers to load right table's tuples
  - May be faster to simply sort the input relations…
  - Results of hybrid merge-join won't be in search-key order

Multi-Column Indexes

- Several situations where multi-column indexes are helpful
- Data warehouses:
  - Standard data warehouse schema design has a few large *fact tables* surrounded by multiple smaller *dimension tables*
  - Relatively small number of records in each dimension table
  - Fact table records are comprised of:
    - A foreign-key reference to a row in each dimension table
    - One or more measures corresponding to that row's set of dimension values
- Queries against such a schema require many joins!



dim_date

dim_time

fact_sales_data
date_id
time_id
region_id
category_id
num_sales
total_revenue

dim_region

dim_category

Bitmap Indexes

- Databases can provide *bitmap indexes* to make queries against these schemas incredibly fast
- A bitmap index on attribute A of a table T:
  - Build a separate bitmap for every distinct value of A
  - The bitmap contains one bit for every record in T
  - For a given value a_j that appears in column A:
    - If tuple t_i holds value a_j for column A, the bitmap for a_j will store a 1 for bit *i*. Otherwise, bit *i* will be 0.
- For such an index to be feasible:
  - Attribute A shouldn't contain too many distinct values (duh)
  - Also, it must be easy to map bit *i* to tuple t_i
  - Specifically, we should generally only <u>add</u> rows to table T

Bitmap Index Example

- An example bitmap index:
  - Sales data warehouse, with bitmap indexes on category and region
- Example query:

  -
    ```
    SELECT SUM(total_revenue)
    FROM fact_sales_data NATURAL JOIN
                dim_region
    WHERE region_name = 'asia';
    ```
  - Could use "region:asia" bitmap; only fetch records with a 1-bit
- But, that's probably not actually faster than just doing a file-scan

Fact table contents:

Date	**Category**	**Region**	**...**
Jun 21	apparel	europe	...
Jun 21	electronics	asia	...
Jun 21	books	asia	...
Jun 22	cookware	n.america	...
Jun 22	books	n.america	...
Jun 23	cookware	asia	...
Jun 23	electronics	europe	...
Jun 23	apparel	asia	...
...	...	...	...

Bitmap indexes:

Category: apparel	1 0 0 0 0 0 0 1 ...
Category: books	0 0 1 0 1 0 0 0 ...
Category: cookware	0 0 0 1 0 1 0 0 ...
Category: electronics	0 1 0 0 0 0 1 0 ...
Region: asia	0 1 1 0 0 1 0 1 ...
Region: europe	1 0 0 0 0 0 1 0 ...
Region: n.america	0 0 0 1 1 0 0 0 ...

Bitmap Index Example (2)

- Reporting queries almost always include multiple conditions:

  -
    ```
    SELECT SUM(total_revenue)
    FROM fact_sales_data NATURAL JOIN
             dim_region NATURAL JOIN
             dim_category
    WHERE region_name = 'asia' AND
               category_name = 'books';
    ```
- Now we can get some real value out of the bitmap indexes!
  - Conjunctive selection predicate: Only include rows that have a 1-bit in <u>all</u> relevant bitmap indexes

Fact table contents:

Date	Category	Region	...
Jun 21	apparel	europe	...
Jun 21	electronics	asia	...
Jun 21	books	asia	...
Jun 22	cookware	n.america	...
Jun 22	books	n.america	...
Jun 23	cookware	asia	...
Jun 23	electronics	europe	...
Jun 23	apparel	asia	...
...	...	...	...

Bitmap indexes:

Category: apparel	1 0 0 0 0 0 0 1 ...
Category: books	0 0 1 0 1 0 0 0 ...
Category: cookware	0 0 0 1 0 1 0 0 ...
Category: electronics	0 1 0 0 0 0 1 0 ...
Region: asia	0 1 1 0 0 1 0 1 ...
Region: europe	1 0 0 0 0 0 1 0 ...
Region: n.america	0 0 0 1 1 0 0 0 ...

Bitmap Index Example (3)

- Our query:

  -
    ```
    SELECT SUM(total_revenue)
    FROM fact_sales_data NATURAL JOIN
             dim_region NATURAL JOIN
             dim_category
    WHERE region_name = 'asia' AND
              category_name = 'books';
    ```
- Compute intersection of relevant bitmap indexes
  - Only retrieve rows that have a 1-bit for all referenced columns
  - This is why it must be easy to find t_i given *i*: don't want to have to access rows with a 0-bit at all

Fact table contents:

Date	**Category**	**Region**	**...**
Jun 21	apparel	europe	...
Jun 21	electronics	asia	...
Jun 21	books	asia	...
Jun 22	cookware	n.america	...
Jun 22	books	n.america	...
Jun 23	cookware	asia	...
Jun 23	electronics	europe	...
Jun 23	apparel	asia	...
...	...	...	...

Relevant bitmap indexes:

Region: asia	0 1 1 0 0 1 0 1 ...
Category: books	0 0 1 0 1 0 0 0 ...
Intersection:	0 0 1 0 0 0 0 0 ...

NULL Attribute Values

- If a row has NULL for the indexed column:
  - Simply store 0 for all bits in corresponding bitmap indexes
- Note:
  - This would be highly unusual in a data warehouse fact-table!
  - Could still occur in other situations

Fact table contents:

Date	Category	Region	...
Jun 21	apparel	europe	...
Jun 21	electronics	asia	...
Jun 21	books	asia	...
Jun 22	cookware	n.america	...
Jun 22	books	n.america	...
Jun 23	cookware	asia	...
Jun 23	electronics	europe	...
Jun 23	apparel	asia	...
Jun 24	NULL	n.america	...
...	...	...	...

Bitmap indexes:

Category: apparel	1 0 0 0 0 0 0 1 0 ...
Category: books	0 0 1 0 1 0 0 0 0 ...
Category: cookware	0 0 0 1 0 1 0 0 0 ...
Category: electronics	0 1 0 0 0 0 1 0 0 ...
Region: asia	0 1 1 0 0 1 0 1 0 ...
Region: europe	1 0 0 0 0 0 1 0 0 ...
Region: n.america	0 0 0 1 1 0 0 0 1 ...

Deleted Rows

- If rows are deleted from table:
  - Still need to easily map bit at index *i* to tuple t_i in the table!
  - Need a way to represent gaps of deleted rows in bitmap index
- Solution: an *existence bitmap*
  - Include an extra bitmap that specifies 1 if row is valid, or 0 if row is deleted
  - Queries that use bitmap index also include existence bitmap in tests

Fact table contents:

Date	**Category**	**Region**	**...**
Jun 21	apparel	europe	...
Jun 21	electronics	asia	...
~~Jun 21~~	~~books~~	~~asia~~	~~...~~
Jun 22	cookware	n.america	...
Jun 22	books	n.america	...
Jun 23	cookware	asia	...
Jun 23	electronics	europe	...
Jun 23	apparel	asia	...
~~Jun 24~~	~~NULL~~	~~n.america~~	~~...~~
...	...	...	...

Bitmap indexes:

Category: apparel	1 0 0 0 0 0 0 1 0 ...
Category: books	0 0 1 0 1 0 0 0 0 ...
Category: cookware	0 0 0 1 0 1 0 0 0 ...
Category: electronics	0 1 0 0 0 0 1 0 0 ...
Region: asia	0 1 1 0 0 1 0 1 0 ...
Region: europe	1 0 0 0 0 0 1 0 0 ...
Region: n.america	0 0 0 1 1 0 0 0 1 ...
Existence	1 1 0 1 1 1 1 1 0 ...

Compressed Bitmaps

- Bitmap indexes aren't *that* large, but they do take up some space
- Bitmap indexes will usually contain large runs of 0- or 1-bits
  - The more distinct values in a given column, the more 0-bits in the corresponding bitmaps
- Very suitable to compression!
- Could use standard compression mechanisms…
  - Would have to decompress before performing bitwise operations

Fact table contents:

Date	**Category**	**Region**	**…**
Jun 21	apparel	europe	…
Jun 21	electronics	asia	…
~~Jun 21~~	~~books~~	~~asia~~	~~…~~
Jun 22	cookware	n.america	…
Jun 22	books	n.america	…
Jun 23	cookware	asia	…
Jun 23	electronics	europe	…
Jun 23	apparel	asia	…
~~Jun 24~~	~~NULL~~	~~n.america~~	~~…~~
…	…	…	…

Bitmap indexes:

Existence	1 1 0 1 1 1 1 1 0 …
Category: apparel	1 0 0 0 0 0 0 1 0 …
Category: books	0 0 1 0 1 0 0 0 0 …
Category: cookware	0 0 0 1 0 1 0 0 0 …
Category: electronics	0 1 0 0 0 0 1 0 0 …
Region: asia	0 1 1 0 0 1 0 1 0 …
Region: europe	1 0 0 0 0 0 1 0 0 …
Region: n.america	0 0 0 1 1 0 0 0 1 …

Compressed Bitmaps (2)

- Several bitmap compression techniques designed to allow efficient bitwise operations on compressed data
  - Doesn’t achieve as high a compression level, but queries don’t incur decompression overhead
- Example: Byte-aligned Bitmap Code (BBC)
  - Bitmap is divided into bytes
  - Bytes containing all 1-bits or 0-bits are “gap bytes”
  - Bytes containing a mixture are called “map bytes”
  - “Control bytes” specify runs of gap-bytes (run-length encoding), and also identify sequences of map-bytes

Compressed Bitmaps (3)

- Byte-aligned Bitmap Code (BBC) achieves very good compression, and is still quite fast…
  - …but CPUs work most optimally with words, not bytes.
- Word-aligned Bitmap Code (WBC) and Word-Aligned Hybrid (WAH) code divide bitmaps into words
  - Doesn't achieve same level of compression as BBC, but is much faster for bitmap operations
  - One research result:
    - WBC/WAH used 50% more space than BBC but was 12x faster
- Other bitmap compression mechanisms as well