Algorithms and Datastructures

Holger Pirk

Slides as of 10/02/22 09:24

About this lecture

Context

- You had two algorithm classes
- You had a database class
- What could possibly be left to learn?
- Well, some of it is really just an application of what you know...
- ...but some is quite specific to data management (most notably joins and aggregations)

Non-Relational Operators

- Sort (Quick-, Merge-, Heap-, Tim-, Radix-, etc.)
- Top-N (using Heaps)

Now, let's do something new…

What is the main problem of database normalization?

Your data ends up all over the place!

Example

Customer

ID	Name	ShippingAddress
1	Holger	180 Queens Gate
2	Sam	32 Vassar Street
3	Peter	180 Queens Gate

Order

ID	CustomerID
1	1
2	2
3	3

OrderedItem

OrderId	BookID
1	1
1	2
2	1
3	3

Book

ID	Title	Author
1	Database Management Systems	Ramakrishnan & Gehrke
2	A Game of Thrones	Martin
3	Distributed Systems	van Steen & Tanenbaum

It needs to be put together again

Enter the Join

Joins are everywhere

- In part due to the whole normalization business
  - These are mostly Foreign-Key joins (we'll talk about those again in the context of indexing)
- In part because combining (joining) data produces value
  - These are more complicated (and interesting)

Examples

- Find users that have bought the same products
- Find the shortest route visting 5 of London's best sights
- Find online advertisements that worked
  - (lead to users searching for a specific term within a timeframe)

Revision

What you should know about joins

- Joins are basically cross products with a selection involving both inputs

Joins

- `select R.r, S.s from R,S where R.id = S.id`
- `select R.r from R,S where R.r = S.s`

Not a join

- `select R.r from R,S where R.r = "something"`
- `select R.r, S.s from R,S where R.r = R.id`

These are all called inner joins

Left, Right and Full Outer Joins

Left Join

A left join $R \overset{\mathsf{L}}{\bowtie} S$ returns every row in R, even if no rows in S match. In such cases where no row in S matches a row from R, the columns of S are filled with NULL values.

Right Join

A right join $R \overset{\mathsf{R}}{\bowtie} S$ returns every row in S, even if no rows in R match. In such cases where no row in R matches a row from S, the columns of R are filled with NULL values.

Left, Right and Full Outer Joins

Full Outer Join

An outer join $R \overset{\mathsf{O}}{\bowtie} S$ returns every row in R, even if no rows in S match, and also returns every row in S even if no row in R matches.

$$R \overset{\mathsf{O}}{\bowtie} S \equiv (R \overset{\mathsf{L}}{\bowtie} S) \cup (R \overset{\mathsf{R}}{\bowtie} S)$$

On matching predicates

The matching function

```
select * from R join S on (R.r = S.s)
```

The matching function need not be equality

- If it is, we call the join an equi-join (these are the most important joins)
  - Algorithmically, they are equivalent to intersections
- If it is an inequality constraint ($<$ or $>$), we call them *inequality joins*

```
select count(*) from event, marker where event.time
between marker.time and marker.time+60
```

- If it is an <> (!= in C syntax), we call it an *anti-join*
- All other joins are called *Theta joins*

Nested Loop Join

Nested Loop Join

Implementation

```
using Table = vector<vector<int>>;
Table left, right;
for(size_t i = 0; i < leftRelationSize; i++) {
  auto leftInput = left[i];
  for(size_t j = 0; j < rightRelationSize; j++) {
    auto rightInput = right[j];
    if(leftInput[leftAttribute] == rightInput[rightAttribute])
      writeToOutput({leftInput, rightInput});
  }
}
```

Nested Loop Join

Example data

R	S
10	8
17	16
7	12
16	1
12	17
8	2
13	7

Nested Loop Join

Properties

- Simple
- Sequential I/O
- Trivial to parallelize (no dependent loop iterations)

Effort

- $\Theta(|left| \times |right|)$
- Can be reduced to $\Theta(\frac{|left| \times |right|}{2})$ if value uniqueness can be assumed
- This is pretty terrible, isn't there something better?
- There is...

The answer. . .

. . . is always either sorting or hashing – my DB professor

Sort-Merge Joins

Sort-Merge Joins

Implementation (assuming values are unique and sorted)

```
auto leftI = 0;
auto rightI = 0;
while (leftI < leftInputSize && rightI < rightInputSize) {
  auto leftInput = left[ leftI];
  auto rightInput = right[ rightI];
  if(leftInput[leftAttribue] < rightInput[rightAttribue])
    leftI++;
  else if(rightInput[rightAttribue] < leftInput[leftAttribue])
    rightI++;
  else {
    writeToOutput({leftInput, rightInput});
    rightI++;
    leftI++;
  }
 }
```

Sort-Merge Joins

Example data

R	S
10	8
17	16
7	12
16	1
12	17
8	2
13	7

Example data

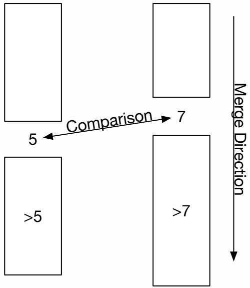
R	S
7	1
8	2
10	7
12	8
13	12

Why Sort-Merge Joins works

Invariants

- Assume, w.l.o.g., that the value on the left is less than the value on the right
- All values succeeding the value on the right are greater than the value on right
- ⇒ No value beyond the value on the right can be a join partner
- ⇒ The value on the left has no join partners succeeding the value on the right
- ⇒ The cursor on the left can be advanced

Visualisation



Sort-Merge Joins

Effort

- $O\left(sort\left(left\right)\right) + O\left(sort\left(right\right)\right) + O\left(merge\right)$, i.e.,
- $O(|left| \times \log|left| + |right| \times \log|right| + |left| + |right|)$
  - Assuming uniqueness

Properties

- Sequential I/O in the merge phase
- Tricky to parallelize
- Works for inequality joins
  - Careful when advancing the cursors

Hash joins

Hash joins

Nomenclature

- We distinguish build-side (the side that is buffered in the hashtable) and probe-side (the one used to look up tuples in the hashtable)

Hash joins

Implementation

```
vector<optional<vector<int>>> hashTable; // <- slots may be empty, hence optional
int hash(int);
int nextSlot(int);

for(size_t i = 0; i < buildSide.size(); i++) {
  auto buildInput = buildSide[i];
  auto hashValue = hash(buildInput[buildAttribute]);
  while(hashTable[hashValue].hasValue)
    hashValue = nextSlot(hashValue);
  hashTable[hashValue] = buildInput;
}

for(size_t i = 0; i < probeSide.size(); i++) {
  auto probeInput = probeSide[i];
  auto hashValue = hash(probeInput[probeAttribute]);
  while(hashTable[hashValue].hasValue &&
        hashTable[hashValue].value[buildAttribute] != probeInput[probeAttribute])
    hashValue = nextSlot(hashValue);
  if(hashTable[hashValue].value[buildAttribute] == probeInput[probeAttribute])
    writeToOutput({hashTable[hashValue].value, probeInput});
}
```

Hash join details... the hash function

Hash-function requirements

Pure no state

Known output domain we need to know the range of generated values

Nice to have

Contiguous output domain we do not want holes in the output domain

Uniform all values should be equally likely

Typical examples

MD5 pretty terrible

Modulo-Division arguably the simplest hash-function

MurmurHash one of the fastest "decent" hash-functions

CRC32 has hardware support

Conflict Handling

When a slot is already filled but there is space in the table...

- We need to put the value somewhere...
- The conflict handling strategy prescribes where

Requirements

- Locality (but not too much :-))
- No holes (probe all slots)

Many exist - let's talk about three

- Linear probing
- Quadratic probing
- Rehashing

Linear Probing

Description

- When a slot is filled, try the next one (distance 1)...
- ...and the next one (distance 2)...
- ...continue until you find one that is free (3,4,5,6, etc.)...
- ...wrap around at the end of the buffer

Advantages

- Simple
- Great access locality

Disadvantages

- Leads to long probe-chains for adversarial input data
- For example, `9,8,7,6,5,4,3,2,2`

Quadratic Probing

Description

- When a slot is filled, try the next one (distance 1)...
- ...double the distance (distance 2)...
- ...continue until you find one that is free (4, 8, 16, etc.)...
- ...wrap around at the end of the buffer
- (note that variants of this principle exist)

Advantages

- Simple
- Good access locality for first probes
  - Increasingly worse after that

Disadvantages

- The first probes still likely to incur conflicts

Rehashing

Description

- Challenge: Distribute probes uniformly
- Solution: Use hashing function for probing as well

Advantages

- Simple
- Conflict probability is a constant

Disadvantages

- Poor access locality
- Challenge: How to make sure all slots are probed
  - Solution: cyclic groups

Hash-join with modulo hashing and linear probing

Simplified Implementation

```
vector<optional<vector<int>>> hashTable;

for(size_t i = 0; i < buildSideSize; i++) {
  auto buildInput = build[i];
  auto hashValue = buildInput[buildAttribute][joinAttribute] % 10; // hash-function
  while(hashTable[hashValue].has_value)
    hashValue = (hashValue++ % 10); // probe function
  hashTable[hashValue] = buildInput;
}

for(size_t i = 0; i < probeSideSize; i++) {
  auto probeInput = probe[i];
  auto hashValue = probeInput[probeAttribute] % 10;
  while(hashTable[hashValue].has_value && //
        hashTable[hashValue].value[joinAttribute] != probeInput[probeAttribute])
    hashValue = (hashValue++ % 10);
  if(hashTable[hashValue].value[joinAttribute] == probeInput[probeAttribute])
    writeToOutput({hashTable[hashValue].value, probeInput});
}
```

Example: Hash-join with modulo hashing and linear probing

Illustration

Example data (linear probing)

```
int hash(int v) { return v % 10; }
int probe(int v) { return (v + 1) % 10; }
probeSide = {7, 8, 10, 12, 13, 16, 17};
buildSide = {1, 2, 7, 8, 12, 16, 17};
```

Hash joins

Properties

- Sequential I/O on the inputs
  - (Pseudo-random access to the hashtable during build and probe)
- Parallelizable over the values on the probe side
- Parallelizing the build is tricky (**Research opportunities!**)

Effort

- $\Theta(|build| + |probe|)$ in the best case
- $O(|build| \times |probe|)$ in the worst case

What did I gloss over here?

Dealing with payloads

What else did I gloss over?

Dealing with duplicate values!

How would you deal with duplicate values?

Hash Joins practicalities

Hashing is expensive

- Especially good hashing
  - Lots of CPU cycles (often more expensive than multiple data accesses)

Slots are often allocated in buckets

- Buckets are slots with space for more than one tuple
- Roughly equivalent to rounding every hash value down to a multiple of the bucket size
- You will sometimes see people implementing buckets as plain linked lists
  - This is called bucket-chaining (what we do is called open addressing)
    - A horrible idea if you care about lookup performance (inserts are okay)

Hash Joins practicalities

Hashtables are arrays too

- They occupy space
- They are usually overallocated by at least a factor two
    - i.e., you allocate twice as many slots as (estimated) tuple inputs

  (obviously adapting the hash-function)
- They are probed randomly in the probe phase (a lot)
    - **You really want to make sure they stay in memory/cache**
- **For this class, assume that, if the hashtable does not fit, every access has a constant penalty**
- Rule of thumb: use Hash Joins when one relation is much smaller than the other

Food for thought: Is that the common case?

What if it that is not my case?

Improving Locality through Partitioning

Partitioning

Fundamental premise:

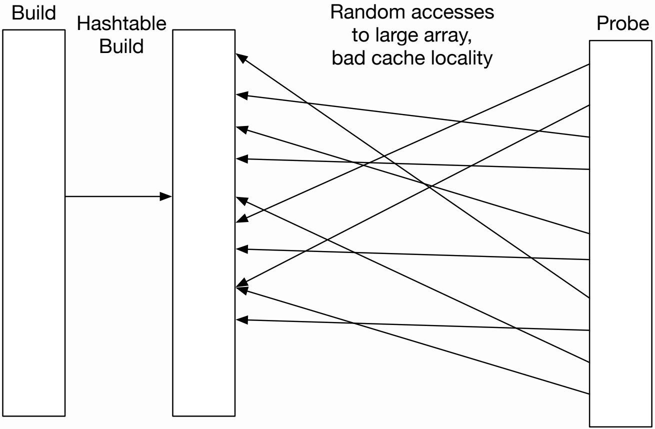
- Sequential access is much cheaper than random access
  - Difference grows with the page size
  - Assume: Random value access cost c
  - Sequential value access cost $\frac{c}{pagesize_{OS}}$

Assume your hashtable does not fit in the buffer page cache/pool

- I.e., if the relation is larger than half the buffer pool
- It can pay off to invest in an extra pass for partitioning

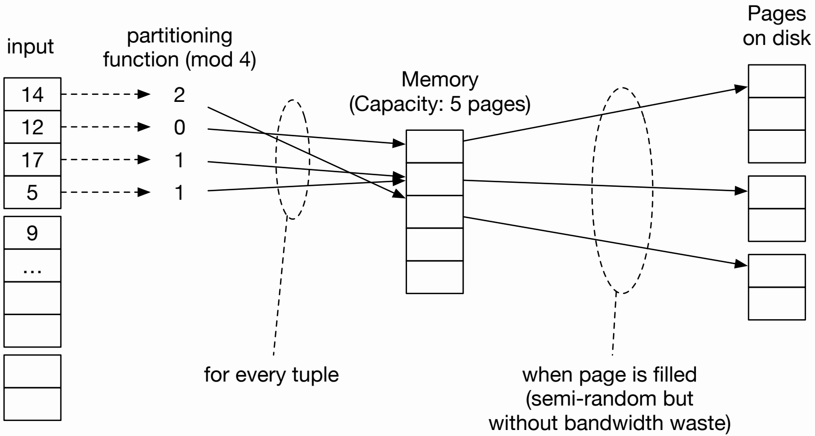
Hashtable thrashing

Visualization



Partitioning - an example

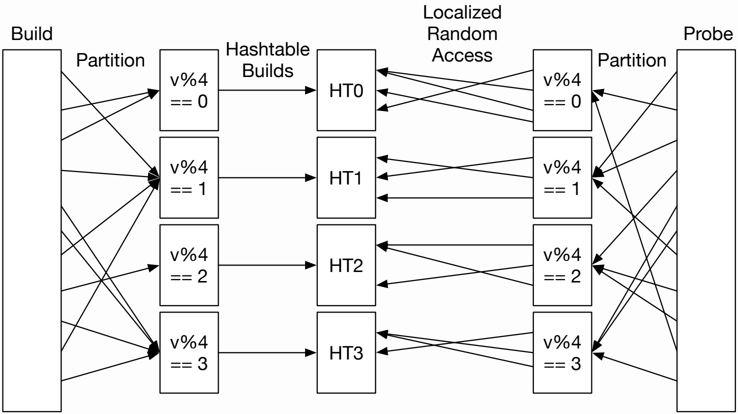
Visualization



Imperial College London

Hashtable probing in partitions

Visualization



Partitioning

Bonus

- You can parallelize the processing of each of the smaller joins
  - because they are disjoint
- You can partition the larger relation as well...
  - ...and only join the overlapping partitions
  - **this is the state of the art in join processing**

Observations

- All of these algorithms have phases:
  - Build & Probe
  - Sort & Merge
- What happens if I store/cache the result of the first phase?
  - I have created an index

Context

Secondary Storage is about replicating data

- The opposite of normalization
  - But in a controlled manner
  - The DBMS is in charge of replicas
  - They can be created and destroyed without breaking the system
  - They are semantically invisible to the user, i.e, results cannot change
  - They can be enormously beneficial for performance

However,

- They occupy space
- *They need to be maintained under updates*
- They stress the query optimizer
- They can only be used for certain operations

Some Nomenclature

Clustered/Primary Index

- An index that is used to store the tuples of a table
- You can have no more than one of these per table
- They may use more space than a table but they don't replicate data (no consistency issues)

Unclustered/Secondary Index

- An index that is used to store pointers to the tuples of a table
- You can have as many as you like per table
- They don't replicate data (some consistency issues)

Our focus is on concepts and data structures...

…not the SQL to create them

That being said...

. . . here is some SQL!

Maintaining indices in SQL

Creating them

```sql
CREATE INDEX index_name ON table_name (column1, column2, ...);
```

Dropping them

```sql
DROP INDEX index_name;
```

This isn't particularly useful yet

- Unclear what kind of index is created
- No control over parameters
- Virtually all systems provide much finer control (look at their documentation)

Creating indices in SQL Server

```
CREATE [NONCLUSTERED]  COLUMNSTORE INDEX ...
CREATE CLUSTERED COLUMNSTORE INDEX ...
CREATE CLUSTERED COLUMNSTORE INDEX with data_compression ...
CREATE UNIQUE CLUSTERED INDEX index_name ...
CREATE UNIQUE  NONCLUSTERED INDEX index_name ...
CREATE CLUSTERED INDEX index_name ...
CREATE NONCLUSTERED INDEX index_name ...
CREATE NONCLUSTERED INDEX index_name WITH FILLFACTOR= ...
...
```

So… what do systems do under the hood?

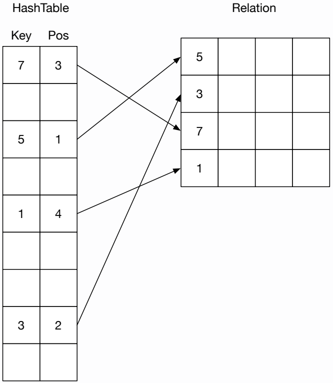
Hash-Indexing

Remember Hash-joins?

- Step one was building a hash-table
- A hash-index is the same thing but persistent
- If you recall: I glossed over payloads
- Now, they are coming back

Hash-Indexing

Unclustered Hash-Index



Clustered Hash-Index

Clustered Relation

Key	Non-Key Attributes		
7	16	9	34
5	12	78	1
1	84	17	69
3	21	55	2

Hash-Indexing

Ephemeral hash-tables

- For hash-joins, we were building one-shot Hashtables
  - there are no new tuples added during query evaluation
    - We knew (roughly) how many tuples are going to end up in the table
  - The hash-table was discarded after the join
  - we did not have to worry about updating it
- *If the hash-table is persistent, all of that changes*

Hash-Indexing

Persistent hash-tables may grow arbitrarily large, so

- Overallocate by a lot
- If fill-factor grows beyond x percent (e.g., 50 percent), **rebuild**
  - **Rebuilds can be very expensive**
  - This leads to nasty load spikes
- Similar for deletes
- Let's talk about those...

Hashtable deletes

- Remember: we used empty slots as markers for the end of probe-chains...
  - and we want short probe chains
- On delete, a value has to be remain in the slot of the deleted value
  - (Food for thought: what happens if we don't)
- Two options
  - Leave the value and mark it as deleted
  - Put another value in there: the *last* value in the probe chain

Here is a proposal:

Hashtable deletes

Deletion strategy (assume uniqueness)

- deleting key `k`
  - Hash k, find k, keep pointer to `k`
  - Continue probing until you find the end of the probe chain
  - If the value at the end of the probe chain has the same hash as `k`, move it into `k`'s slot
  - Otherwise, mark `k` as deleted
    - (fill `k`'s slot with the next value that hashes into the probe chain)
- Example: delete 23 first, delete 14 next

Illustration

Clustered Relation

Key	Non-Key Attributes			Deleted
9	16	9	34	
27	5	61	45	
12	12	78	1	
23	84	17	69	
5	45	71	20	
17	9	42	83	
14	21	55	2	

Bottom line: It is complicated!

Usefulness of Hash-Indices

- Remember: we said, hashjoins are good for equi-joins
  - Because hash-tables allow the quick lookup of a specific key
- Not useful for inequality-joins
  - Because hash-tables do not allow to find the adjacent values

Usefulness of Hash-Indices

- The same applies here:
  - Persistent Hash-tables are great for hash-joins and aggregations (duh!)
  - (assuming they are built on the join/aggregation key columns)
- They also help a lot to reduce the number of candidates if not all columns are indexed (on equality selections):
  - `select * from customer where name = "holger"`
- Not great for anything else:
  - `select * from customer where id between 5 and 8`

Bitvectors

Definition

A sequence of 1-bit values indicating a boolean condition holding for the elements of a sequence of values

- E.g., $BV_{==7}([4, 7, 11, 7, 7, 11, 4, 7]) = [0, 1, 0, 1, 1, 0, 0, 1]$
- CPUs don't work well with individual bits – they work in CPU words
  - for simplicity let's assume a word is 8-bit (in practice it is at least 32 bit)
- $BV_{==7}([4, 7, 11, 7, 7, 11, 4, 7]) =$
  $128 * 0 + 64 * 1 + 32 * 0 + 16 * 1 + 8 * 1 + 4 * 0 + 2 * 0 + 1 * 1 = 89$

Bitmap Indices

Definition

A collection of bitvectors on a column (one for each distinct value in that column)

- Useful if there are few distinct values in a column
- Usually, the bitvectors are disjoint
  - I.e., In every position/row, exactly one value is set to one

Bitmap Indexing

Visualization

Column	Bitmap Index ==5	Bitmap Index ==3	Bitmap Index ==9
5	1	0	0
3	0	1	0
9	0	0	1
9	0	0	1
9	0	0	1
5	1	0	0
3	0	1	0
9	0	0	1

Using Bitmaps

```
unsigned char** bitmaps; // a collection of bitmaps on column
void scanBitmap(byte* column, size_t inputSize, byte value) {
  unsigned char* scannedBitmap = bitmapForValue(bitmaps, value);
  for(size_t i = 0; i < inputSize / 8; i++) { // iterate over bitmap
    if(scannedBitmap[i] != 0) {
      unsigned char bitmapMask = 127; // binary 10000000
      for(size_t j = 0; j < 8; j++) {
        if((bitmapMask & scannedBitmap[i]) && column[i * 8 + j] == value)
          writeOutput(column[i * 8 + j]);
        bitmapMask >>= 1;
      }
    }
  }
}
```

Bitmap Indexing – Usefulness

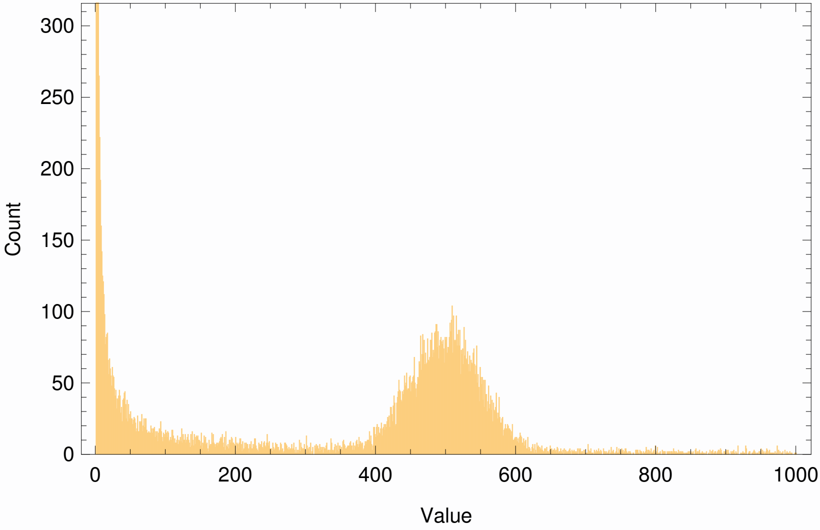
- Bitmaps reduce bandwidth need for scanning a column
  - in the order of the size of the type of the column in bits
- Predicates can be combined using logical operators on bitvectors
- Arbitrary (boolean) conditions can be indexed by some systems
  - $BV_{>7,<12}([4,7,11,7,7,11,4,7]) =$
    $128*0+64*1+32*1+16*1+8*1+4*1+2*0+1*1=125$
- Special form: binned bitmaps

Binned Bitmaps

- Idea: Have n bitvectors, each with a predicate covering a different part of the value domain
- For example (assuming our column type is `byte`),
  - Bin 1: 0 through 7
  - Bin 2: 8 through 20
  - Bin 3: 20 to 255
- Make sure the conditions span the entire value domain
- Problem: Index cannot distinguish values in a bin (unless bin contains only one value)
  - Can only produce candidates
  - False positives need to be eliminated

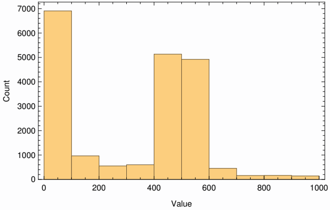
Binning

Let's say our value distribution look like this...



Binning strategy: Equi-Width

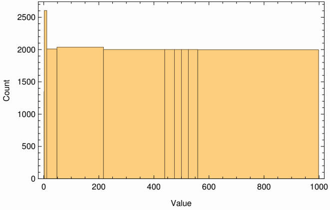
Illustration



- Simple to configure:
  - Bin width: $\frac{(\max(column) - \min(column))}{numberOfBins}$
- Limited use when indexing non-uniformly distributed data
  - Many false positives in highly populated bins
  - For example, 34% of values need to be validated when checking for value 99, 99.5% of which are false positives

Binning

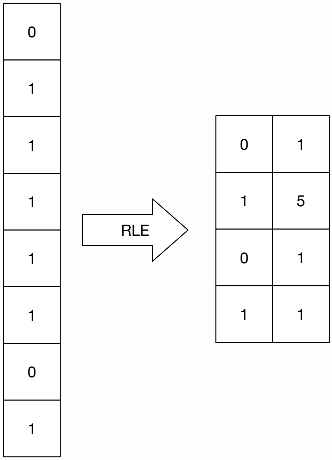
Illustration



- Resilient against non-uniformly distributed data
  - False-positive rate independent is value independent
- Bin construction is tricky:
  - Basically: sort values and determine quantiles
  - Usually performed on sample
- Distributions may change over time (which requires re-binning)

Run-Length-Encoding (for bitmaps)

Visualisation

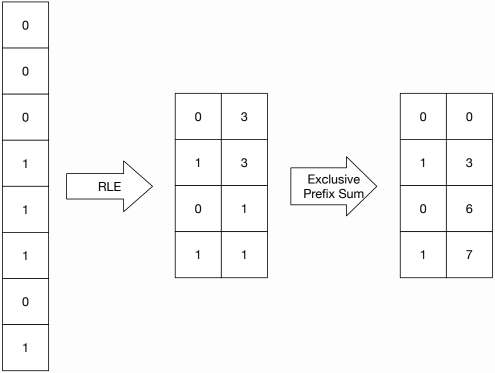


- Sequentially traverse the vector
- Replace every run of consecutive equal values with
  - a tuple containing the value (*Run*) and the number of tuples (*length*)
- Works really well on high-locality data
- Requires sequential scan to find value at a specific position

Imperial College London

Run-Length-Encoding with Length Prefix Summing

Visualisation



- Replaces scan with binary search

All of these have a problem: limited updatability

Let's discuss some indices that are updatable!

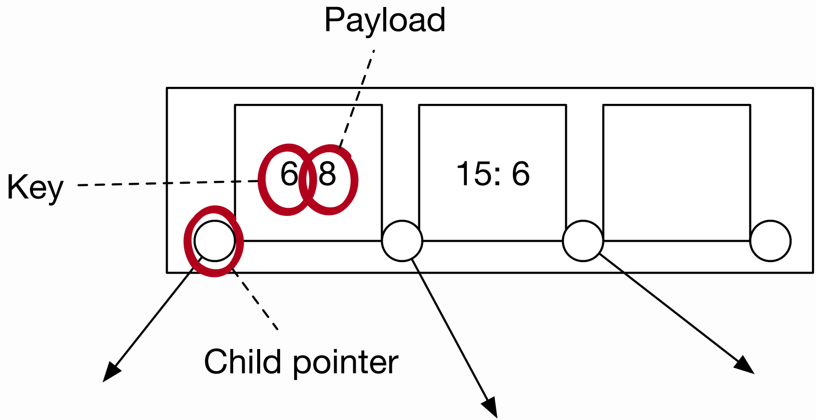
B-Trees

Basic Idea

- Databases are I/O bound (on disk)
  - $\rightarrow$ Minimize the number of page I/O operations
- There are many equality lookups
- There are also many updates
  - Hash-tables have nasty load-spikes on update
- Solution: Use a tree
- You know many binary trees: R/B-Trees, AVL, etc.
- Database trees use high-fanout trees to minimize page I/Os

B-Trees: The nodes

Anatomy of a B-Tree Node (n=4)

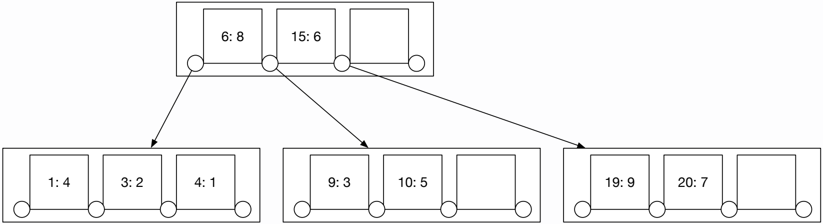


B-Trees

Definition

- A **balanced tree** with out-degree n (i.e., every node has $n-1$ keys) and the following property
- The **root** has at least one element
- Each **non-root node** contains at least $\lfloor \frac{n-1}{2} \rfloor$ key/value pairs

Example (n=4)



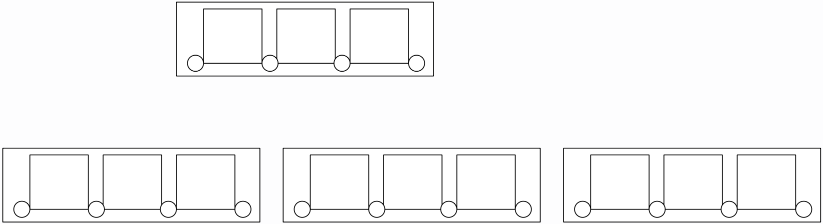
Maintaining balanced B-Trees

Insert

- Find the right **leaf-node** to insert (walk the tree) and insert the value
- If the node overflows, split the node in two halves
- Insert a new split element (the one in the middle of the split-node) in the parent
- If the parent overflows, repeat the procedure on the parent node
    - If the parent is the root, introduce a new root

Maintaining balanced B-Trees

Example



Maintaining balanced B-Trees

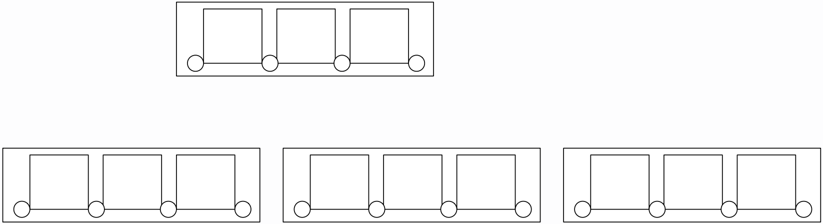
Under delete

- Find the value to delete
  - if it is in a leaf node, delete it
  - if it is in an internal node, replace it with the maximum leaf-node value from the left child (removing the value from the leaf-node)
- If the affected leaf node underflows, rebalance the tree bottom up
  - Try to obtain an element from a neighbouring node, make it the new splitting key an move the splitting key into the node (be done on success)
  - On failure, the neighbouring node cannot be more than half-full and can be merged with this one
  - merge and remove the parent spliting key
  - If parent underflows, rebalance from that one (bottom up)

Imperial College London

Maintaining balanced B-Trees under delete

Example



Problems with B-trees

Access properties

- They can support range (`between 5 and 17`) scans but
  - it is complicated (need to go up and down the tree)
  - it causes many node traversals
  - Node sizes are usually co-designed with page sizes
  - Node traversals translate into page faults - we want to keep those to a minimum

Implementation complexity

- Two kinds of node layouts or space waste
  - Leaf pointers aren't used
  - Most of the data lives in leaf nodes

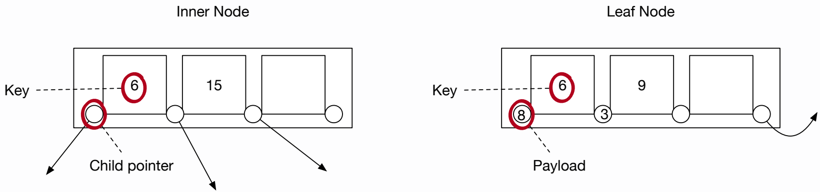
$\mathrm{B}^+ - Trees$

Idea

- Make range scans fast by
  - keeping data only in the leafs (no up and down)
  - linking one leaf to the next
  - inner-node split values are replicas of leaf-node values
- Only have a single kind of node layout...
  - ...with different interpretation of the fields

$\mathrm{B}^{+} - Trees : Nodes$

Anatomy of a $\mathrm{B}^{+} - TreeNode(n = 4)$

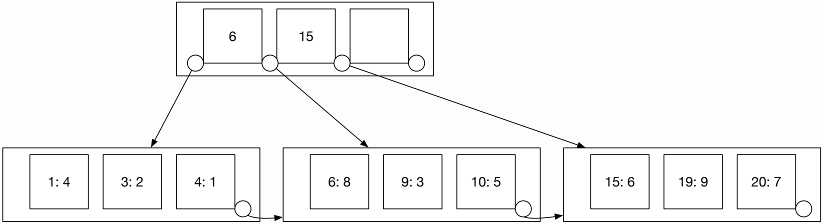


B^{+} $- Trees$

Definition

- Almost the same structure as B-Trees but
  - All data is stored in the leaf nodes
  - Inner nodes only contain copies of values from leaf-nodes
  - Every leaf node (except the last contains a pointer to the next leaf node)

Example



$\mathsf{B}^+ - Trees$

Balancing

- Largely the same
- Deletes of inner-node split values imply replacement with new value from leaf node

Foreign-Key Indices

In SQL

```
ALTER TABLE Orders ADD FOREIGN KEY (BookID_index) REFERENCES Book(ID);
```

- Foreign Key (FK) constraints specify that
  - for every value that occurs in an attribute of a table
  - there is **exactly** one value in the Primary Key (PK) column of another table
- The DBMS needs to ensure that the constraint holds
  - On insert/update, the DBMS needs to look up the primary key value
  - Instead of storing the value, the DBMS could store a pointer to the referenced Primary Key or tuple

Foreign-Key Indices

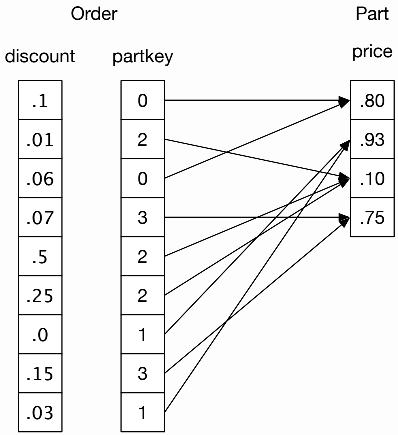
Order

Discount	Partkey
.1	17
.01	4
.06	17
.07	9
.5	4
.25	4
.0	29
.15	9
.03	29

Part

Key	Price
17	.8
29	.93
4	.10
9	.75

Index (in DSM)



Use of Foreign-Key Indices

- The PK/FK constraint implies the number of join partners for every tuple: 1
- Resolving the FK reference column directly yields the join partner tuples
  - FK indices are basically pre-calculated joins
- Not of much use for anything else
  - However, many joins are PK/FK joins (because they stem from normalization)

Use of Foreign-Key Indices

- Foreign-Key Indices have very few downsides
  - Cause insignificant extra work under updates
  - Do not cost significant space (a pointer per tuple)
  - No extra query optimization effort: if they can be used, they should be
- SQL-Server does not implement them

Thank you

Provide feedback, please!

https://co572.pages.doc.ic.ac.uk/feedback/algorithmsandindices

Get the slides online

https://co572.pages.doc.ic.ac.uk/decks/AlgorithmsAndIndices.pdf