Advanced Databases Documentation

Release 1.0

m

February 23, 2015

1 Introduction to Indexing **3**
1.1 File Sorting . . . 3
1.2 Hash Table Indexing . . . 5
1.3 B+ trees . . . 5
1.4 Queries . . . 7

2 Introduction to Spatial Indexing **9**
2.1 Spatial Data Types . . . 9
2.2 Integration of Spatial Types into Databases . . . 10
2.3 Types of Queries for Points . . . 11
2.4 Grid Indexing . . . 12
2.5 Z-Ordering . . . 13
2.6 K-d Tree . . . 15
2.7 Quad Tree . . . 19
2.8 R-Tree . . . 21

3 Big Data **29**
3.1 Concepts . . . 29
3.2 Google File System (GFS) . . . 30
3.3 MapReduce . . . 35
3.4 Bigtable . . . 39
3.5 Percolator [Perc2010] . . . 41

4 Indices and tables **45**

Bibliography **47**

PDF (ebook) version of the slides: pdf

CHAPTER

ONE

INTRODUCTION TO INDEXING

This document provides introductory material to indexing concepts for disk based access.

GOAL

Find a piece of data **fast**, even though the data is on **disk**

The fact that data is disk has serious implications for performance. why?

Concept

Physical Data Management in Databases:

Databases store **records**. Each record is a fixed length **item** that contains multiple data primitives. The structure of a record is usually specified by a **schema**.

Databases love fixed length records. Again, this has implications because of disk storage. Why are fixed length records preferable?

Example

Relation ski(brand str(10), name str(10), lenght int, rocker bool)

Assume chars take a byte, int takes 4 bytes, bool takes a byte (why not a bit?). Each record requires 25 bytes. 3 skis take 75 bytes.

The database creates a file for the data in each relation. Here is how the file might look for 3 skis:

k2_____Apache____00B20Dynastar__Cham_____00B71Atomic____Arc_____00CB0

The integers are 178, 183, and 203 in hex.

- How do I find all skis of length 183?
- How many records do I need to examine to find all skis of length 183 if there are 1,000,000 skis?

1.1 File Sorting

The data file is essentially an array of records such that the structure of each record is known. The easiest way to find something in an array is to search it! A good way to search an array is to sort it, then use a binary search. There is only 1 problem:

- The file storing a relation can be huge! much larger than memory

We cannot use a traditional sorting technique. Why?

The solution is to use an external memory sort. We will use an external merge sort. The basic question is "How can we sort :math"X bytes of data in Y bytes of memory"?

To do this, we want to maximize disk performance by playing to its strengths, and maximize memory performance by playing to its strengths.

Disk strengths are

1. sequential read
2. reading large chunks of data (divisible by the page size)

Memory strengths (that we want to take advantage of):

1. random access
2. reading small chunks of data (at least a memory word)
3. taking advantage of cache

Here is the basic algorithm:

External Merge Sort Algorithm. Sort X bytes of data in Y bytes of memory.

1. create a set F of X/Y files (be careful about the remainder) For each file $f \in F$, read Y bytes of data into memory, sort it, and write to f

That is the sort step, it creates a bunch of sorted files.

2. Create $(X/Y) + 1$ input buffers, and one output file named f_o Each buffer will hold data from 1 file in F. Fill each buffer with data from its corresponding file Until all files are empty, find the smallest value at the front of an input buffer, remove it, put it in the output buffer.

- If an input buffer empties, fill it with data from its associated file.
- If the output buffer fills, write it out to the output file and clear the buffer.

That is the merge step

Exercise

What is the time complexity and space complexity of merge sort?

So, the approach to indexing a file using a sorted file is to sort, then simply do a binary search

---

Example

The original data file above is sorted on ski length. How do I find the record containing the 178cm ski?

Problems:

- What if I want to search the above file for ski names?
- What if I add new skis to the database?
- What if I update skis, remove skis?
- What if 10000 users are trying to read the data, and I am making lots of inserts?

One of the major problems with sort indexing is that the file itself is the index. The file and index are integrated! This has a lot of not-so-nice implications and restrictions.

1.2 Hash Table Indexing

Lets remove the index from the actual data. lets have 1 data file, and 1 index file. We need a way to associate an record with something in the index file (we don't want to duplicate a potentially huge relation)

Concept

Key. A key is a unique identifier for a record in a relation. Usually, it isi desirable for the key to be much smaller than the record in terms of data size. The key and record form a key/value pair.

Relations come with a built in concept of a key, a **primary key** for the relation. So, we simply need a hash table that takes a key, and returns what? (not a record)

For our previous example, assume ski names are unique. We want to search based on ski names. Instead of sorting the whole file, just create a hash table **H**. Remember, a new tuple starts at multiples of 25:

```
H['Cham']   -> 25
H['Apache'] -> 0
H['Arc']    -> 50
```

Exercise

One common criticism of hash tables is that they wast some space because you always need a little extra room in a hash table to control the number of collisions. It turns out that the wasted space is actually pretty small. In your language of choice, write a simple hash table that hashes integers to locations in an array. The simple recipe for a good hash function is here : http://www.cs.siue.edu/~marmcke/docs/cs340/hashing.html#good-hash-functions.

Once you have the hash table, try inserting objects into a hash table of length 4 until you get a collision. How many insertions did you get before a collision occurred? repeat 10 times.

Increase the size of the hash table to 100, and repeat the above experiment. You should find that you generally get a pretty full table before things start colliding.

You should find that in practice, with a properly designed hash function, the criticism is unfounded since the extra space required is actually quite small. In fact, hash tables are one of the most useful data structures out there, and have made possible much of the big data movement (as we will see).

Issues:

1. What do I do if I want to search on Brand?
2. What is the time complexity of making/using a hash table?
3. Hash tables are usually memory stuctures. What are the implications of storing them disk? (linked lists)
4. What if we have a perfect hashing function?

Hash table indexes are nice, but we want something a little more predictable.

1.3 B+ trees

For a quick review of basic disk concepts, and more information on B+ trees, see the following: http://www.cs.siue.edu/~marmcke/docs/cs340/trees.html#b-trees

B trees and (more commonly used) B+ trees are the answer to the problems with hashing. They provide:

1. An index structure that is seperate from the data
2. Hard lower and upper bounds on size and time to complete operations

3. Design specific to disk based structures.

B trees and their variants are used in virtually every DBMS out there.

Again, we need a concept of a **key** that uniquely defines a record. We will store complete records in one file (the relation). We will create a new file to hold the index. The index will contain keys that identify records, and the **disk address** of that record in the relation's file. This means we don't need to re-order the relation's data file ever, we can simply append to it. Also, we can create as many index files as we want for whatever we want to search on.

We will talk about B+ trees, they are easier to use and manage, and work more naturally.

B+Tree

A B+tree contains 2 types of nodes: **internal** nodes and **leaf** nodes.

- internal nodes contain keys and pointers to other internal nodes or leaf nodes.
- leaf nodes contain keys and disk addresses of records in the relation's data file.

A node is **fixed size**. Usually, a node is the size of a disk page. Sometimes it will be a multiple of the size of a disk page. This means the number of items stored in a node is dependent on the size of the item, and the size of the disk page (usually 4k).

let b be the *branching factor* of nodes in a B+tree. b indicates the number of child nodes for a node in the tree. Let m be the actual number of keys in a node. For internal nodes $\lceil b/2 \rceil \leq m \leq b$. For leaf nodes: $\lfloor b/2 \rfloor \leq m \leq b - 1$. Exceptions are made for the root node if there are simply aren't enough items in the tree to reach the minimum threshhold.

Node Structure: Each node contains the following structure: [pointer, key, pointer, key, pointer, ..., key]. Pointers in internal nodes point to other nodes. Pointers in leaf nodes contain disk addresses, expect for the last pointer which points to the next successive node to the "right" in the tree

Note in B+ trees, keys are *repeated* in internal nodes and in leaves. This means **every** key will show up in a leaf. Some keys will show up in internal nodes. This is a major difference with standard B Trees

Searching the tree. Search the tree like a binary search tree

Insertion Into the Tree

1. Find the node at which to insert the item
2. If there is room in the node, insert it.
3. If there is no room, find the median value x.
4. Create two new nodes, place all keys $k \leq x$ in the left node, and all keys $k > x$ in the right node.
5. Move the median value x up to the parent node of the split node, and connect the tree. If there is no room for the median value in the parent node, repeat steps 4-5 on that node

B and B+trees are *complete trees*. They do not grow at the leaves, instead, they grow from the *root*!

Example

Put the following keys into a B Tree and a B+Tree. Let the branching factor $b = 4$:

20 5 8 40 23 31 1 10 33 41 22 7 16 44 27 35 11 9 2 17

How many nodes are visited to find the value 20? 33? 44?

How many nodes are visted to traverse the tree starting with value 31?

what is the time complexity of search and insert?

The algorithm for searching and inserting into a B+tree is straightforward, but the implementation of insertion can be tricky. In the worst case, a split can propogate up the entire tree to the root. It is useful to keep track of a stack of node pointers that lead back from the node of insertion ot the root.

As with most trees, the time complexities of the tree are dependent upon th tree height, for B+trees, the tree is **always** balanced, but trees with more full nodes might be shorter than trees holding the same amount of data with less full nodes.

For example, a B+tree with branching factor b and height h can have :

- at most b^h pointers to data records. why?
- at min $\lceil b/2 \rceil^h$ pointers to data records. why?
- a height bound for n elements of $O(\lg_b n)$, meaning logarithmic search, insert and delete!

Example

Assume a 4096 (4Kb) disk page and records with integer keys (4 bytes). Assume a 64 bit system (8 byte memory pointers). If B+tree blocks contain no header information, the we have $(4096 - 8)/(4 + 8) = 340$ keys per block (1 pointer and 1 key takes 12 bytes. the -8 is for the last pointer).

Lets assume a typical block will have 255 keys (somewhere between 340 and 340/2=170).

- With a root, we can store 255 pointers to records
- With a root and 255 leaves, we have pointers to $255^2 = 65,025$ records
- with a root and 255 internal nodes and 255 leaves for each internal node, we have pointers to $255^3 = 16.6mil$ records

So, such a B+tree with height 3 requires 3 disk accesses to search 16.6 mil records! Furthermore, the top few levels of a B+tre are typically buffered in memory, eliminating the need to disk access in may cases.

1.4 Queries

The whole point of indexing is to speed up queries (since disk is slow). In an RDBMS, queries are usually **SELECT FROM WHERE** queries. Here are some common query types:

1. Simple selection (SELECT firstName FROM studentTable WHERE lastName like 'Mc%')
2. Range (SELECT firstName FROM studentTable WHERE lastName > 'Mc' AND lastName < 'N')
3. Joins (Select className FROM studentTable S, classTable C WHERE lastName like 'Mc%' and S.id == C.id)

CHAPTER

TWO

INTRODUCTION TO SPATIAL INDEXING

Traditional indexing works extremely well for traditional data. Spatial data, however, does not fit the expected model of trafditional data that RDBMSs like to use.

2.1 Spatial Data Types

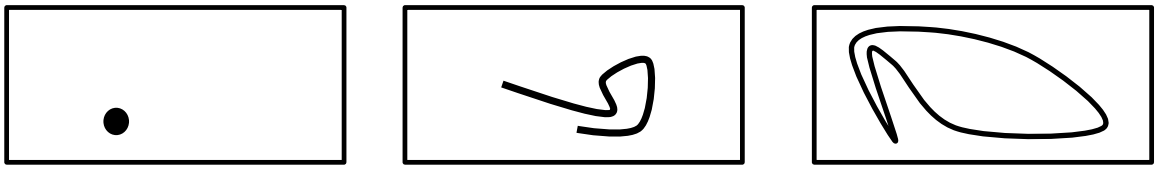
There are three general spatial data types that are typically used to represent geographic reality. We will focus on two dimensional types. The three types are: **point, line, and region**. Furthermore, they are differentiated into two classes: **simple** and **complex**.

Definition: Simple point, line, and region

A simple point is a single point in euclidean space: (x,y)

A simple line is a connected, one dimensional object in euclidean space with two end points.

A simple region is a single, simple, minimal cycle separating the interior of the region from the exterior of the region (A Jordan curve)

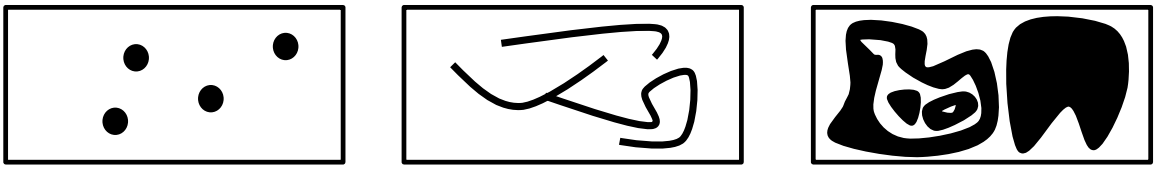


Definition: Complex point, line, and region

A complex point, sometimes called a multi-point, is a collection of non-overlapping, simple points

A complex line is a collection of disjoint simple lines

A complex region is a collection of faces and holes. A face is a simple region, possibly containing holes. A hole fits the definition of a simple region, but the interior of a region lies on the exterior of a hole, and the exterior of a region lies on the interior of a hole. Faces can contain multiple holes, and multiple faces can occur within a hole. Faces and holes can touch along their boundaries at a finite number of discrete points.



In mathematical terms, all data types are considered to be regular open point sets. This has desirable properties for the semantics of operations. All spatial objects have three components: an interior, and exterior, and a boundary.

- Points have only a boundary and an extrior.
- Lines have a boundary defined as the end points of the line. The interior is all parts of the line not at the end points. The exterior is everything else (the rest of the embedding space)
- Regions have a boundary that seperates the interior and exterior of the region.

Why are simple data types not sufficient? (think in terms of geographic reality, operations that must be performed, and object oriented conceptualization).

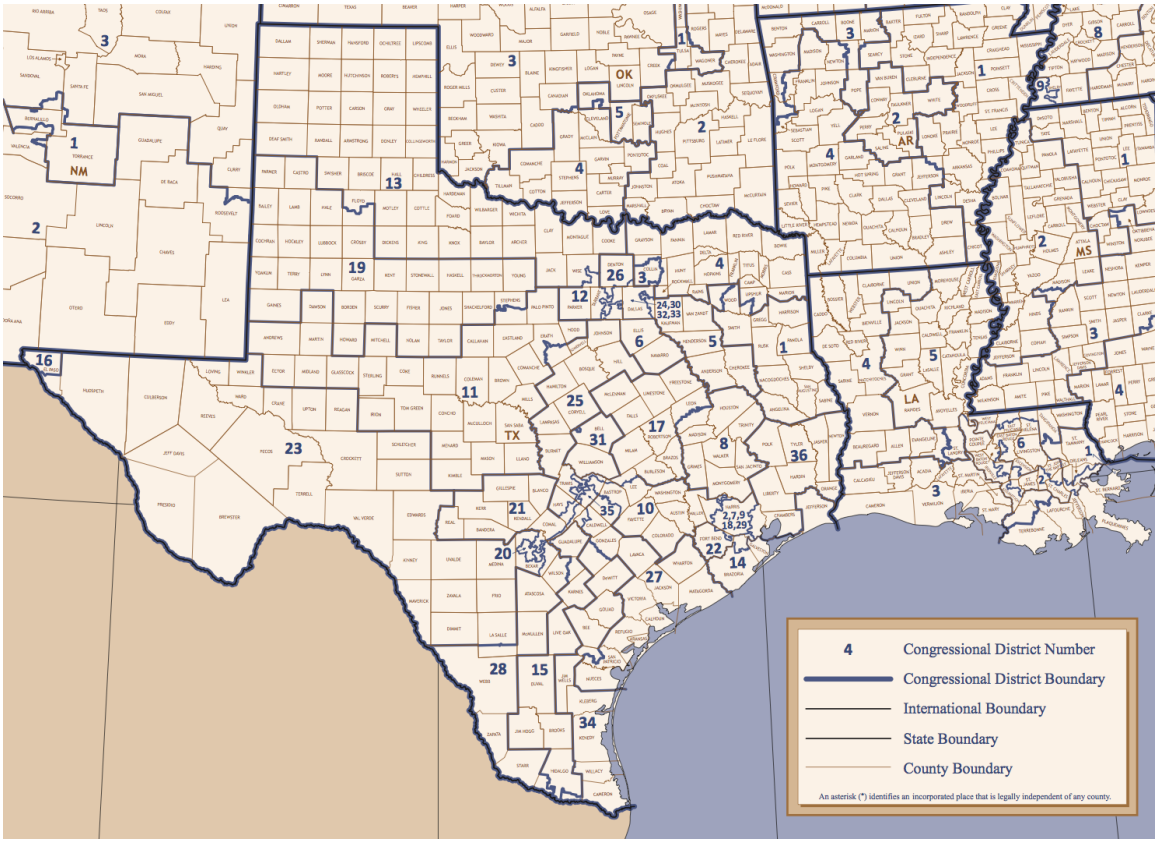
2.2 Integration of Spatial Types into Databases

The mathematical definitions of spatial types are based on point set topology. Types are defined by using infinite point sets, not directly representable in computers.

So, we discretize them by storing their boundaries. Furthermore, boundaries are stored as straight line segments. Straight line segments are easy, they have easy geometry, they require no computation to generate, and they are accurate enough.

- GPS is accurate to about 5 meters (with a good signal).
- 1 meter (approximately within the US and similar lat/lons) is approximately 0.00001 degrees lat/lon. 6 digits of precision fits nicely into a floating point value on computers!
- Wars tend to be fought over rivers, not 5 meters of land. (Georgia/Tenessee water wars!)

So... how many straight line segments does it take to represent Texas, Louisiana, Arkansa? Should we shoot for compact representations, or more detailed representation. what about portions of boundaries that are straight? Take a look at the TIGER data (The image below is published by the US. Census Bureau at http://www.census.gov/geo/maps-data/maps/reference-cd113.html) ...



Databases love fixed size records. How do we incorporate spatial data

---

Concept: Spatial Data In RDBMS

RDBMSs like fixed sized records, in fact, many traditional index structures rely on this.

Solution: In a table, create a column of a geometry type. The record simply contains a pointer (disk address) of the actual geometry. 1 level of indirection.

Counties (Name string(30), stateAbbrv string(2), population int, geometry region)

Now, records get to use the existing index structures, but we have the ability to add new index types for the spatial data.

2.3 Types of Queries for Points

When considering index structures, it make sense to consider the types of queries that will access them. Recall the B+ tree's ability to link adjacent leaf nodes supporting range queries. For point data, there are lots of possible types of queries, here are some common ones:

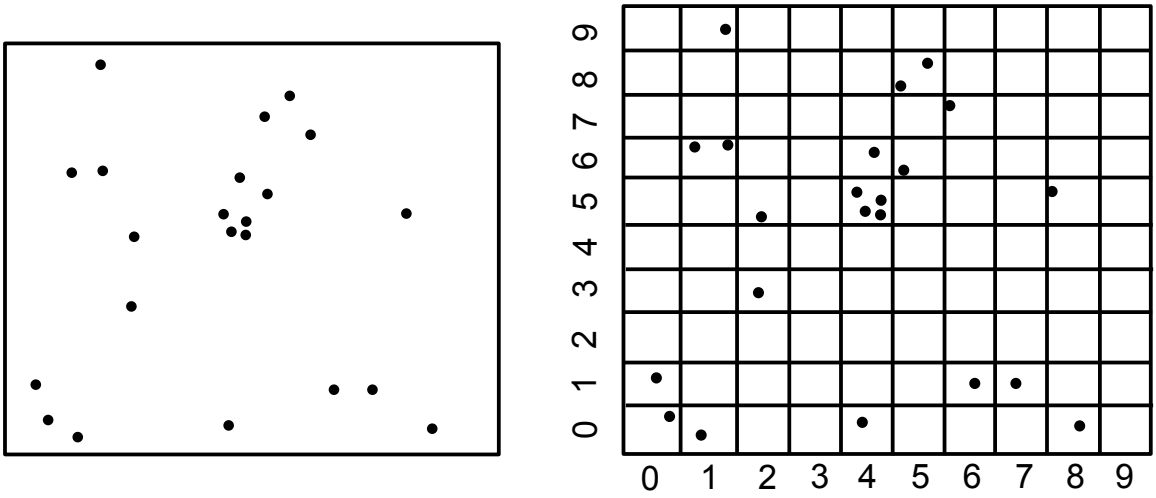
- KNN: *Find the k nearest neighbors to a query point.* The value of k can be 1 to find the nearest neighbor, in which cas it is just a nearest neighbor query.
- Range: *Find the points that lie in a query window*. Sometimes called a window query, box query, etc. The user supplies the rectangle defining the window.
- Selection: *Find all points equal to a query point* or *find all points within 2 miles of a query point*. Typically used in conjucntion with a predicate or operation (ie., **equal** or **withinRadius**).

What are some examples of real world applications that use these query types? Consider ecology, biology, mapping, climatology, anything else you can think of.

2.4 Grid Indexing

We typically know something about the data. For example, how much to expect, roughly where it will be, some kinds of bounds.

Simplist way to index space just break it up into a uniform grid.



Computational based. We don't have to store the grid cells. For a universe from 0-10 in X and Y direction, and 10 cells in each dimension, what cell does the point $(5.5, 7.5)$ lie in? $(cell_x, cell_y)$ where $cell_x = 0 + x/((10 - 0)/10)$.

Where do we store the actual points (remember our disk page behavior)? Each grid cell must point to a disk page where the points in that cell are stored. If a page fills up, we simply chain them, much like the leaves of a B+tree.

So, we need an index file that translates grid cells to disk pages.

Queries

What is the procedure to compute a nearest neighbor query for the point $(5.1, 6.5)$? In the BEST case, how many disk pages do we read? In the worst case, how many disk pages do we read? How many times do we query the index file? Repeat the question for $(8.5, 8.5)$.

What is the procedure to compute a range query: "Find all points in the rectangle defined by $(4, 4), (6, 7)$. Best case disk page reads? worst case disk page reads?

What is the procedure to find all points in within a radius of 2 from $(5.1, 6.5)$? Best case disk page reads? worst case disk page reads?

One problem for grid files, the disk pages for each cell are not necessarily stored near each other on disk!

Advantages:

1. simple
2. can utilize a hash table to make the actual index file smaller
3. If the grid is densly packed, you don't have the problem of empty cells!

Disadvantages:

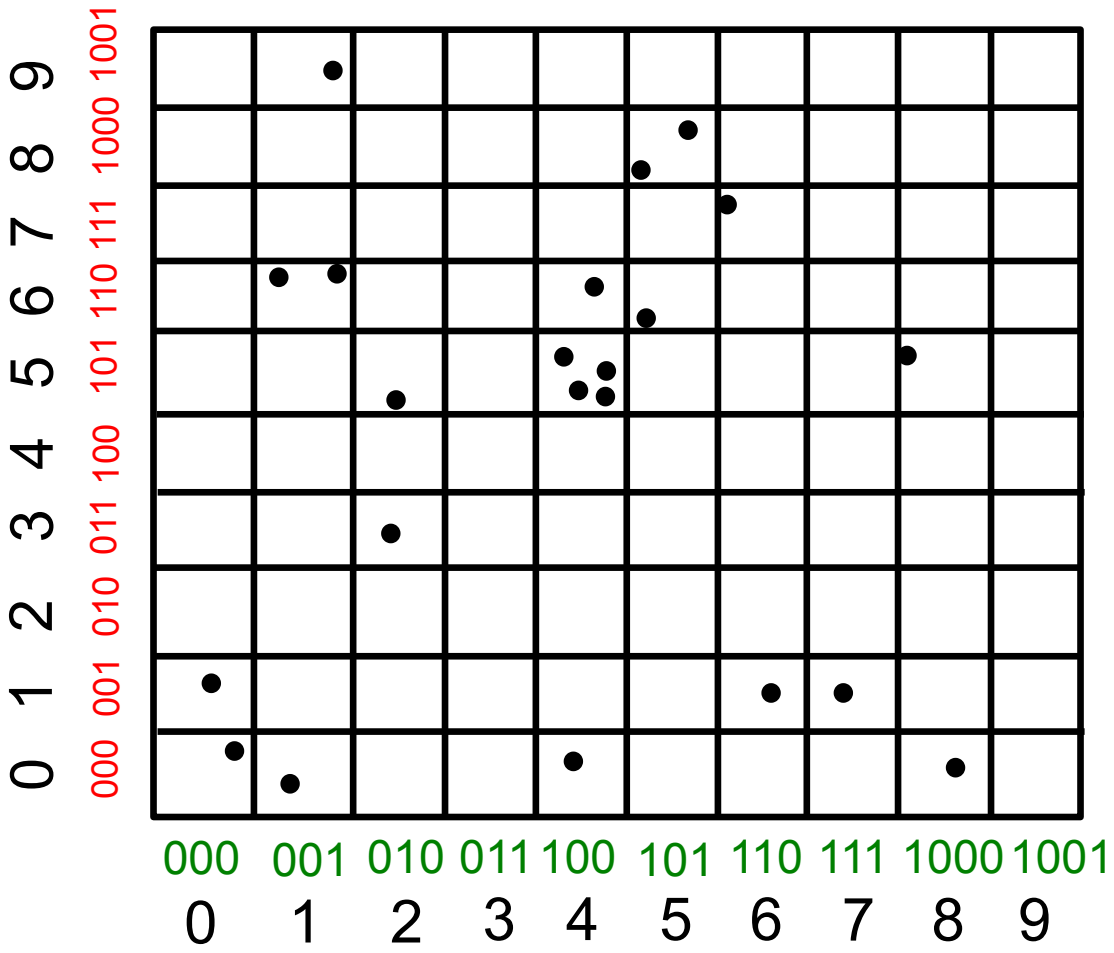
1. sparsely packed grids must indicate that cells are empty somehow, it might mean $O(n^2)$ space in the index file. Possibly list in the index file that the cell is empty. Possibly use a hash table approach.
2. Does not scale to regions particularly effectively.
3. How do we store cells consecutively on disk? We can't! disk is a one dimensional structure, we are storing a 2 dimensional structure!

2.5 Z-Ordering

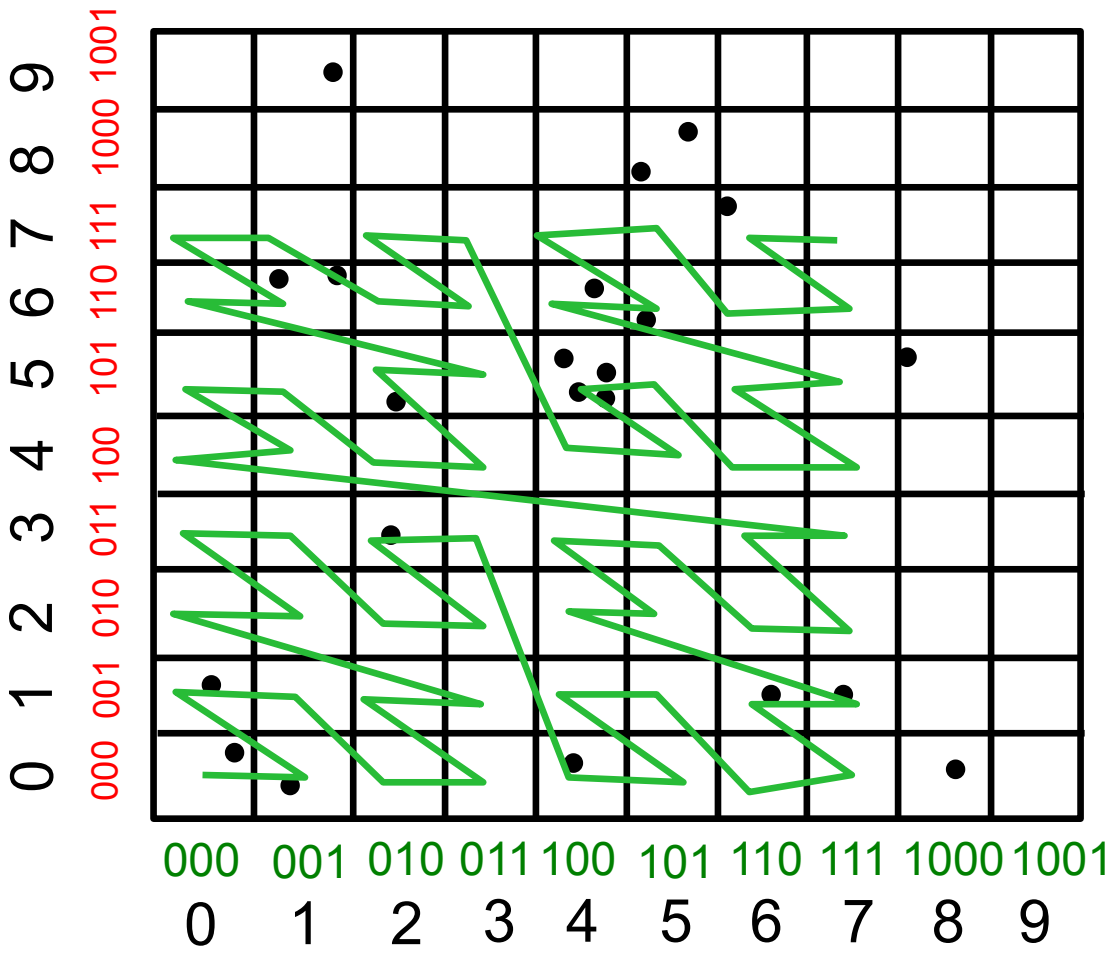
Clearly, 2 adjacent grid cells in a grid index cannot always be adjacent on disk (a cell has 8 neighbors!). How do we arrange cells on disk to improve seek time and reduce effective rotational latency? Instead of random cell placement on disk, follow some kind of ordering.

Z ordering is a method of mapping 2 (or higher) dimensions to a lower dimensional ordering (typically 1). We want a two dimensional cell structure to map to a 1 dimensional disk addressing scheme such that cell neighbors tend to end up close to each other on disk.

Z ordering works by interleaving the bit representations of a cell address:



If we interleave the bits of each cell address, we end up with a linear ordering of the cells. You can see where the Z ordering gets its name:



For example, the cells (0,0), (0,1), (1,0), (1,1) get the following interleaved bit addresses (red bit goes first and are numbers, green bits are **o** and **t**):

0 o 0 o 0 o 0 o: (0,0) = 0

0 o 0 o 0 o 0 t: (0,1) = 1

0 o 0 o 0 o 1 o: (1,0) = 2

0 o 0 o 0 o 1 t: (1,1) = 3

0 o 1 o 0 t 1 o: (2,5) = 38

0 o 1 o 0 t 1 t: (3,5) = 39

Queries:

What is the procedure to compute a nearest neighbor query for the point $(3.5, 4.5)$? In the BEST case, how many disk pages do we read? In the worst case, how many disk pages do we read? How many times do we query the index file?

What is the procedure to compute a range query: "Find all points in the rectangle defined by $(4, 4), (6, 7)$. Best case disk page reads? worst case disk page reads?

What is the procedure to find all points in within a radius of 2 from $(4, 6)$? Best case disk page reads? worst case disk page reads?

So... why is the z-ordering superior to the straight grid file? Spatial Proximity

2.6 K-d Tree

The problem with grid based stuctures is that they are static! They do not adapt effectively to the input. In other words, they are highly sensitive to data skew, in which a lot of the data might exist in a small area of the grid. If the data is evenly distributed, grids work well.

The k-d tree is an adaptive data structure that will adjust to fit the data. It is a **binary tree** in which every node represents a point in one of **k** dimensions. Non leaf nodes effectively divide a hyperplane into two half-spaces (two parts). Each level in the tree typically cycles among which axis the hyperplane will be perpendicular to. Items that are left of the hyperplane will go into a left subtree, items right of a hyperplane will go in a right subtree.

Construction of the tree is as follows:

1. the axis used to generate splitting hyperplanes are cycled repeatedly
2. node points are selected by taking the median of the points being placed in the subtree

---

Example

Lets build a k-d tree with the points: (1,6), (2,3), (3,2) (4,5), (5,4), (6,1). We will start splitting on the x dimension.

A split will stop when it enocounters a previosly existing split line.

Root node: split on x axis at median of x values = 4: (4,5)

left subtree will have: (1,6), (2,3), (3,2)

split on y axis at median of y values: 3: (2,3)

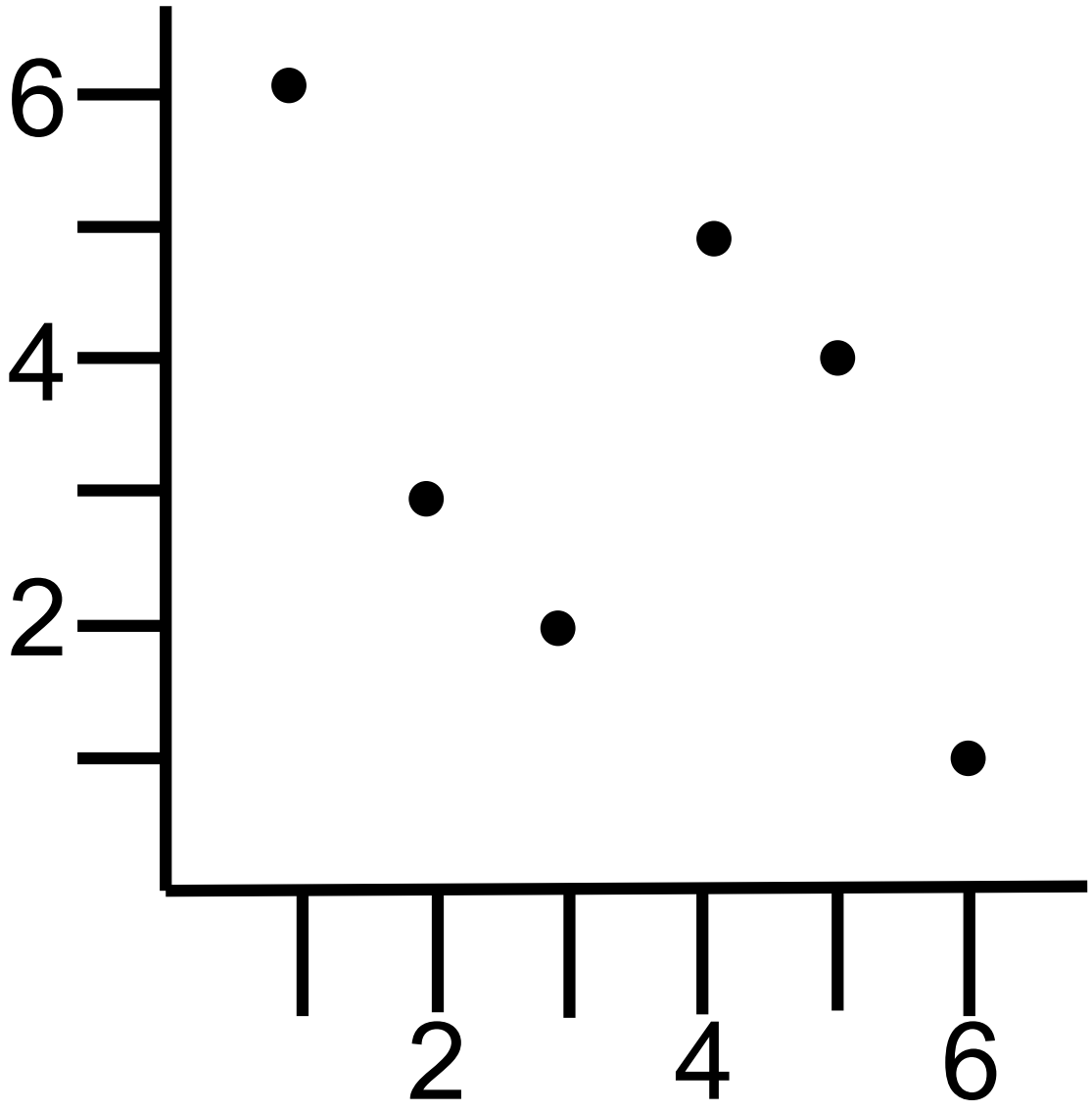
left subtree will have: (3,2) – make final split on x axis

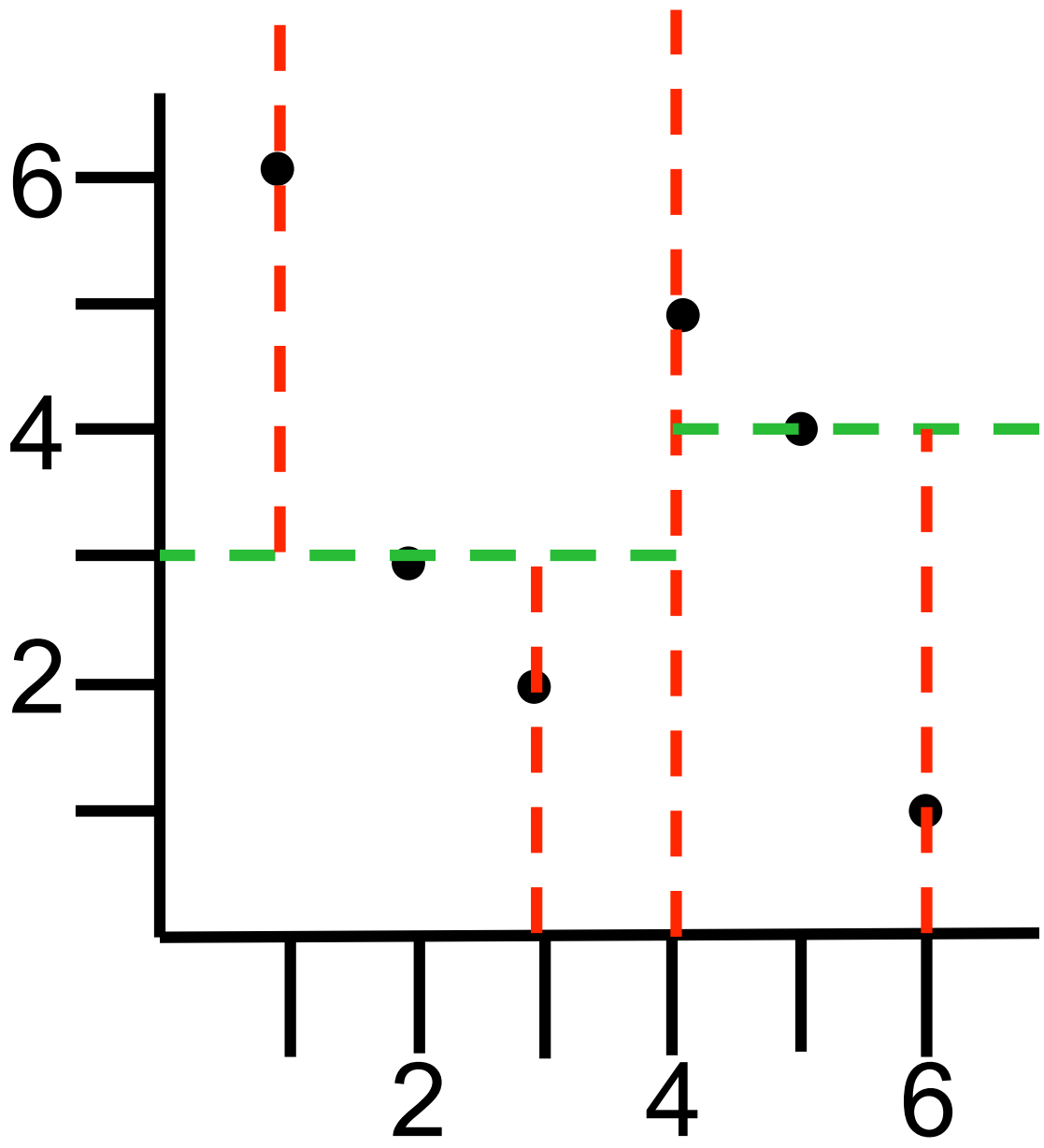
right subtree will have: (1,6) – make final split on y axis

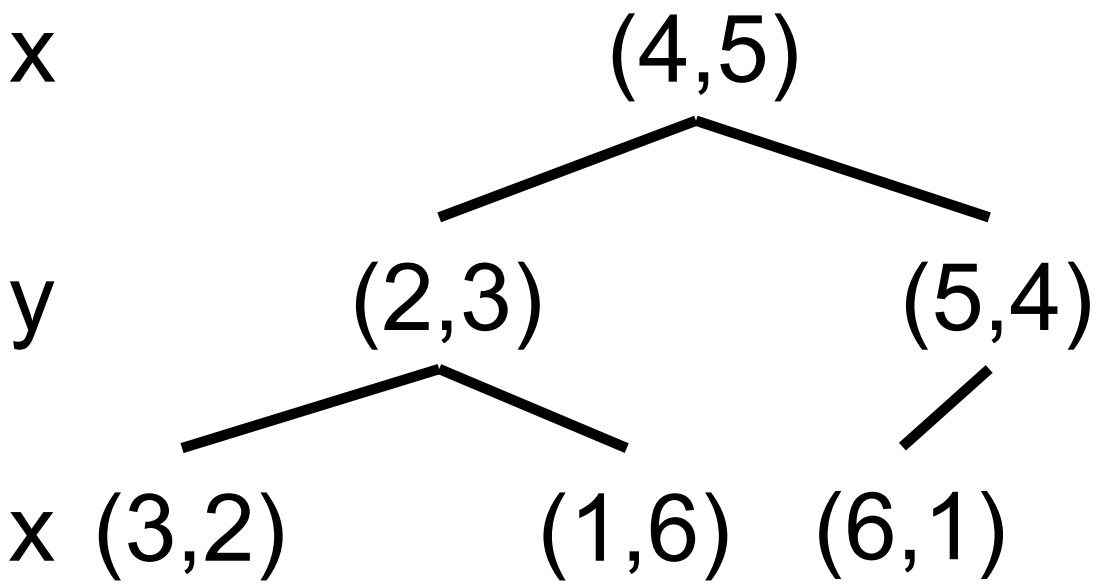
right subtee will have: (5,4), (6,1)

split on y axis of median of y values 4: (5,4)

left subtree will have: (6,1) – make final split on x axis


6
4
2
2
4
6


6
4
2
2
4
6



Note that the X levels in the tree act like the oprdering of a binary search tree, as do the Y levels. This property allows us to discover the rectangular subdivision of space implied by the tree rather easily. For example, to find the rectangle contianing the point (3.5,3.5), we simply follow the tree like a binary search tree, keeping track of the relevant extrema values:

- less than (4,5) on an x split, so the rectangle (l,r,b,t) = (l,4,b,t)
- greater than (2,3) on the y split, so (l,4,3,t)
- greater than (1,6) on the x split, so (1,4,3,t)

Therefore, the contianing rectangle ranges are 1 to 4 in the x direction, and 3 to +infinity in the y direction

Inserting: Inserting follows the same binary search tree procedure to identify the insertion point. Insertions occur at leaves.

Deleting: find the node to delete, and then rebuild the tree consisting of all children of that node. OR, do a binary search tree style delete

Properties

Using the tree creation algorithm, a K-d tree is balanced (we always choose the median value). Inserts and deletes may cause the tree to become unbalanced. Thus, it is not a self-balancing search tree.

Time complexity of building the tree:

1. To find the median value of a particular dimension, we scan all n input points.
2. We can then arrange all points to be on the appropriate side of the median point.
3. for each side, we repeat at the next level, again scanning all n input points.
4. This process is repeated $lg\ n$ times because it forms a balanced binary tree.

Because k is an input parameter, it affects the number of comparisons to splitting planes.

Final time complexity is $O(kn\ lg\ n)$ for building the tree.

The basic k-d tree cannot be re-balanced, but adaptive k-d trees do exist.

2.6.1 Nearest Neighbor Search

Basic approach to the nearest neighbor:

1. Find the insertion point if we were to insert the point. Record the parent of that node as the closest.
2. Walk back up the tree, if a node is closer, keep that one. If a closer node could exist on the other side of the splitting plane, check that sub tree.

Example

Find the nearest neighbor in the above tree to (4.1, 2)

Search the tree to identify node (6,1) with distance = $\sqrt{(1.9^2 - 1^2)} = 1.6$

back up to (5,4) with distance = 1.7 ((6,1) is still closest), $2 + 1.6 = 3.6, 3.6 < 4$, so no need to search the right child of (5,4)

back up to (4,5) with distance = 2.9 ((6,1) is still closes), $4.1 - 1.6 = 2.5 < 4$, so we need to search the left child of (4,5)

recursively search. Find the insert point as a child of (3,2). with distance = 1.05 **closest!**

back up to (2,3) with distance = 1.8 ((3,2) is still closest). $3 + 1.8 = 4.8, 4.8 > 3$, so we need to check right child of (2,3)

find (1,6), which is farther than 1.05, so back up the tree

already checked both sides of (2,3)

already checked both sides of (4,5)

2.7 Quad Tree

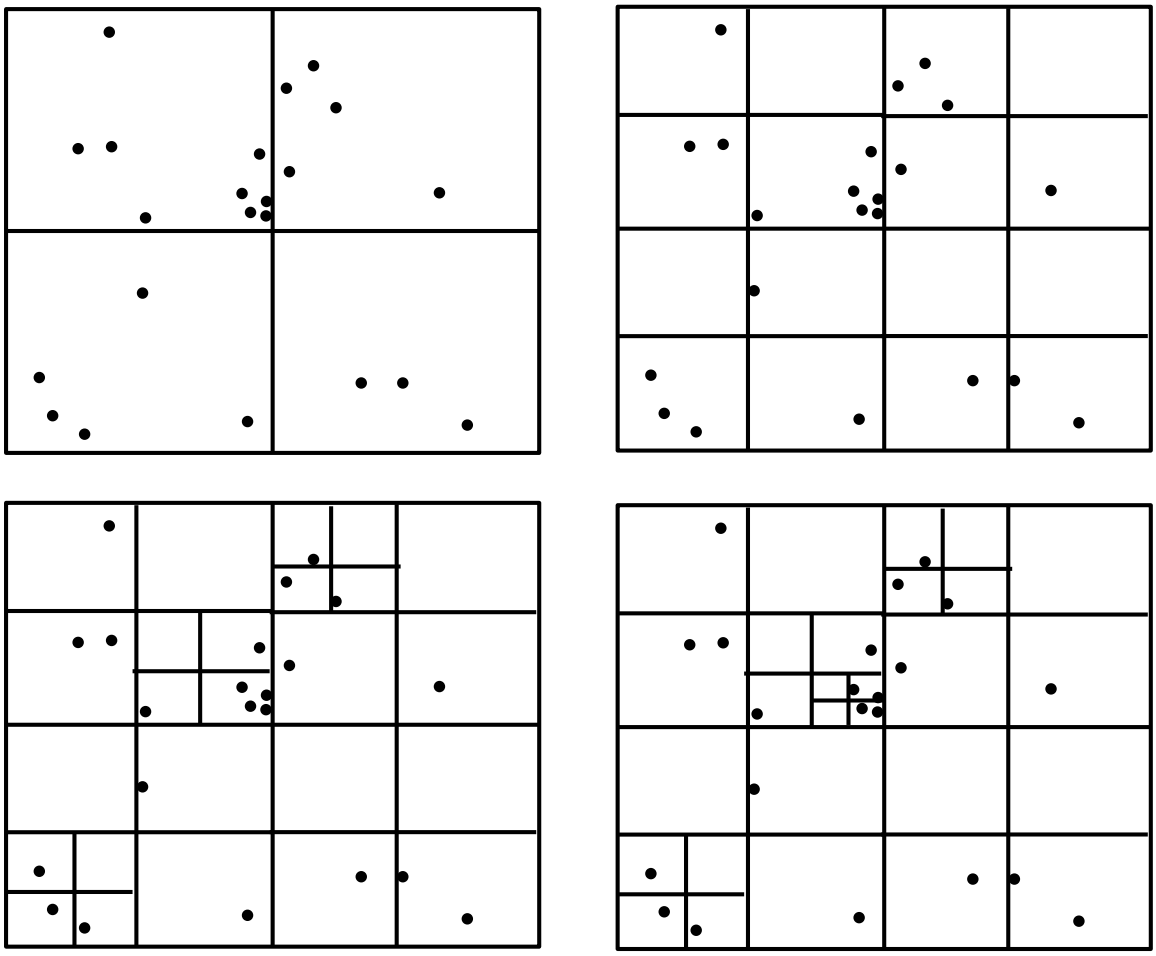
Quad trees are another popular alternative to the fixed spatial decomposition scheme of grid files. Quad trees recursively divide space into quadrants until each quadrant contains a maximum number of elements. The maximum number of elements is a decision made before the quad tree is constructed, and will often depend on a disk page size. In essence, the maximum number of elements in a quadrant should be manageable based on some designe constraint.

Let m be the maximum number of elements grouped together into a leaf node of a quad tree for point data.

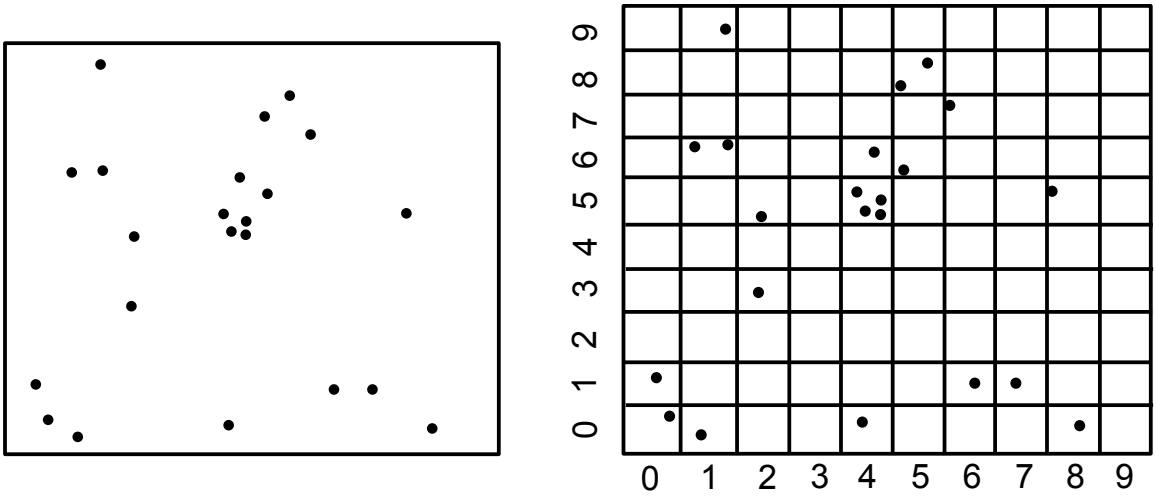
The basic idea is to begin with the entire embedding space, and see if the are more than m elements. If so, then the space is divided into 4 equal quadrants. A root node is created, and leaf nodes, one per quadrant, are attached. Points in each quadrant are recorded in the corresponding leaf node. The procedure is applied recursively to each leaf node. If a leaf node contains fewer then m points, the we do not subdivide the node..

Example

Create a quad tree using the following data set..



Compare the spatial decomposition to that of the uniform grid:



Avantages:

- It is an adaptive spatial decomposition scheme. It adapts to the data, much like a k-d tree.
- A node can hold multiple data elements, unlike the basic k-d tree.

- The tree structure is readily visible by a visualization of the quadrants. Much more so than a k-d tree.

Disadvantages:

- It is limited in use as a disk based index structure because the internal nodes are nowhere near the size of a disk page. Schemes can be achieved that pack internal nodes into disk pages, but this is less efficient than a structure that is naturally aligned with disk-page-sized nodes.

Queries

What is the procedure to compute a nearest neighbor query for the point $(5.1, 6.5)$? In the BEST case, how many disk pages do we read? In the worst case, how many disk pages do we read? How many times do we query the index file? Repeat the question for $(8.5, 8.5)$.

What is the procedure to compute a range query: "Find all points in the rectangle defined by $(4, 4), (6, 7)$. Best case disk page reads? worst case disk page reads?

What is the procedure to find all points in within a radius of 2 from $(5.1, 6.5)$? Best case disk page reads? worst case disk page reads?

2.8 R-Tree

Quad trees intuitive, they are adaptable to the data distribution, and they provide simple query constructions, but they do not generalize to a disk based structure very effectively. R-trees are the answer to that drawback of quad trees. The idea of an R-tree is to generalize a B-tree to handle two (or multi-) dimensional data.

The first problem we run into is that the problem of dimensionality with spatial data. B-trees rely on a total ordering or integers that corresponds to physical nearness across the dimensionality of the data (one dimensional integers). Points, are two dimensional. We can define a total ordering of points, but two points that are adjacent in ordering are not necessarily physically near to each other. The ordering does not imply physical nearness.

Exercise

Define a total ordering over points. Draw three points where two points that adjacent in the ordering are not the nearest points in the embedding space. Do the same with regions.

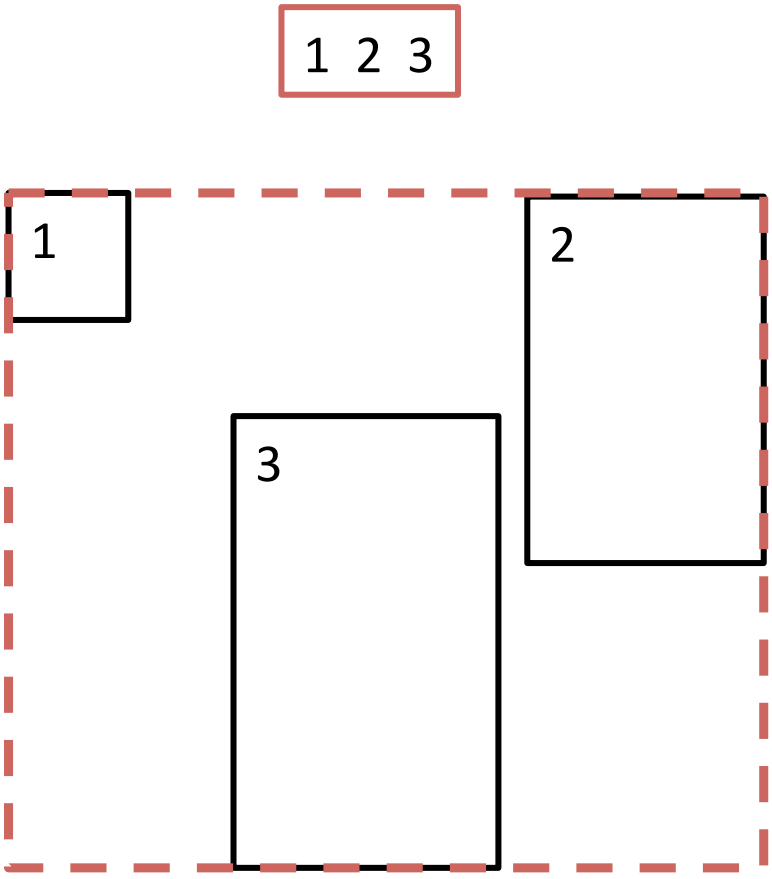
The idea of the R-tree is that physically near objects are grouped together in nodes in the R-tree. Like a B-tree, actual objects are stored only at leafs, and internal nodes provide a search path to the appropriate leaves. R-trees are balanced, have a fixed node size, and have constraints as to the minimum amount of objects allowed in a node. In essence, R-trees are similar in concept to B-trees, but with modifications to handle multi-dimensional data.

The main problem with R-trees is making sure that internal nodes don't overlap too much, internal nodes don't contain too much empty space, and that objects that are close together in space are actually stored close together in the tree.

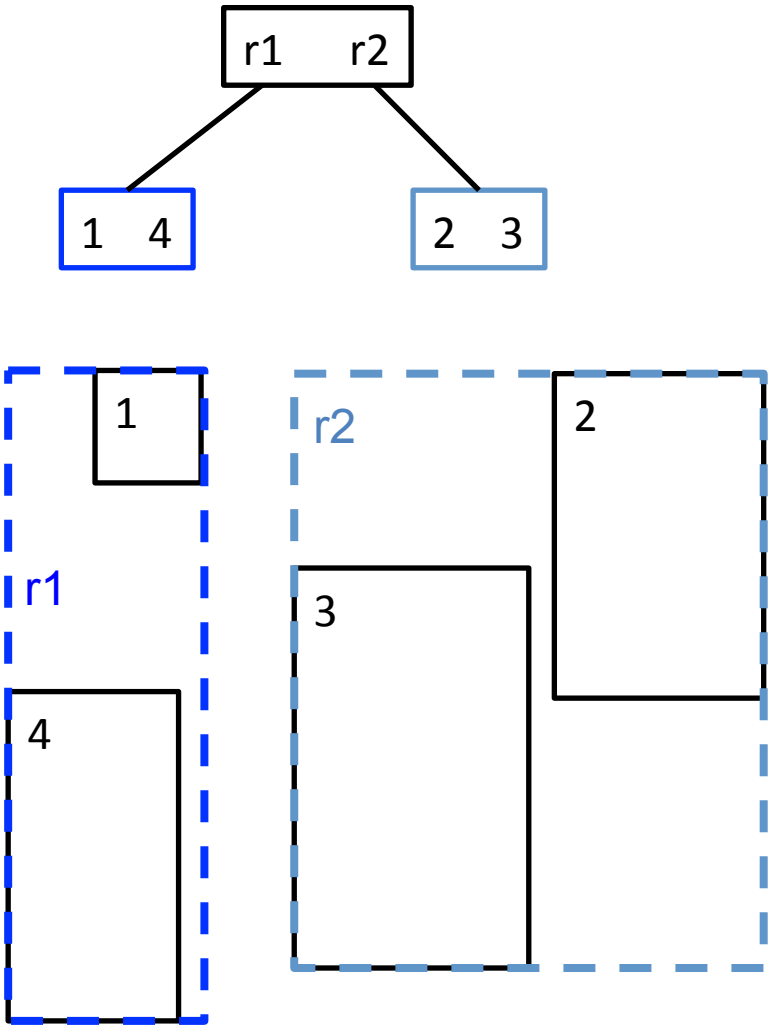
An internal tree node will contain minimum bounding rectangles (MBRs). Associated with each MBR is a **single** pointer to a node such that the MBR minimally encloses all spatial objects in that node. (Note the single pointer, as opposed to a data element separating two pointers in a B-tree). When a node is full, it is split, objects are assigned to new nodes, and the MBRs of each node are adjusted. A MBR is then promoted (like a B-tree) to the parent node. A split can propogate all the way to the root, possibly growing the tree by one level.

Example

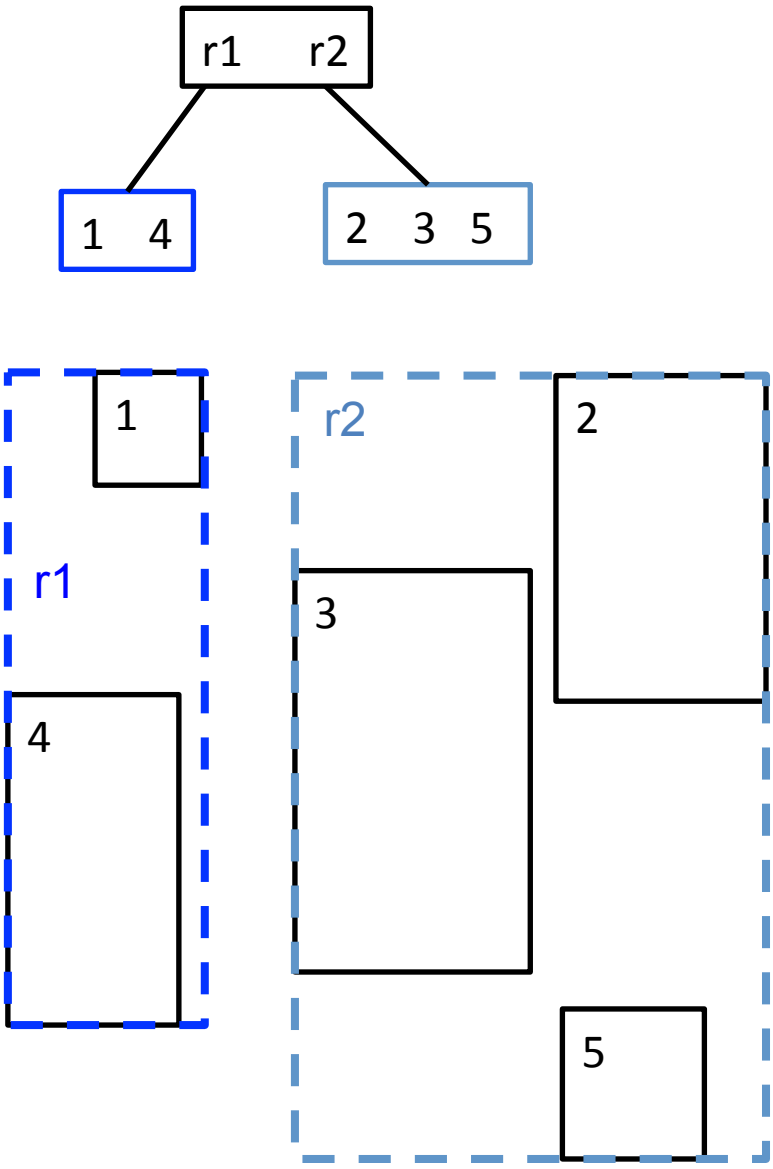
In this example, we will build up the R tree by adding a single regions at a time. Assume an R-tree that holds at most 3 object pointers per node. Adding regions to the root node is straightforward, and behaves much like a B tree.



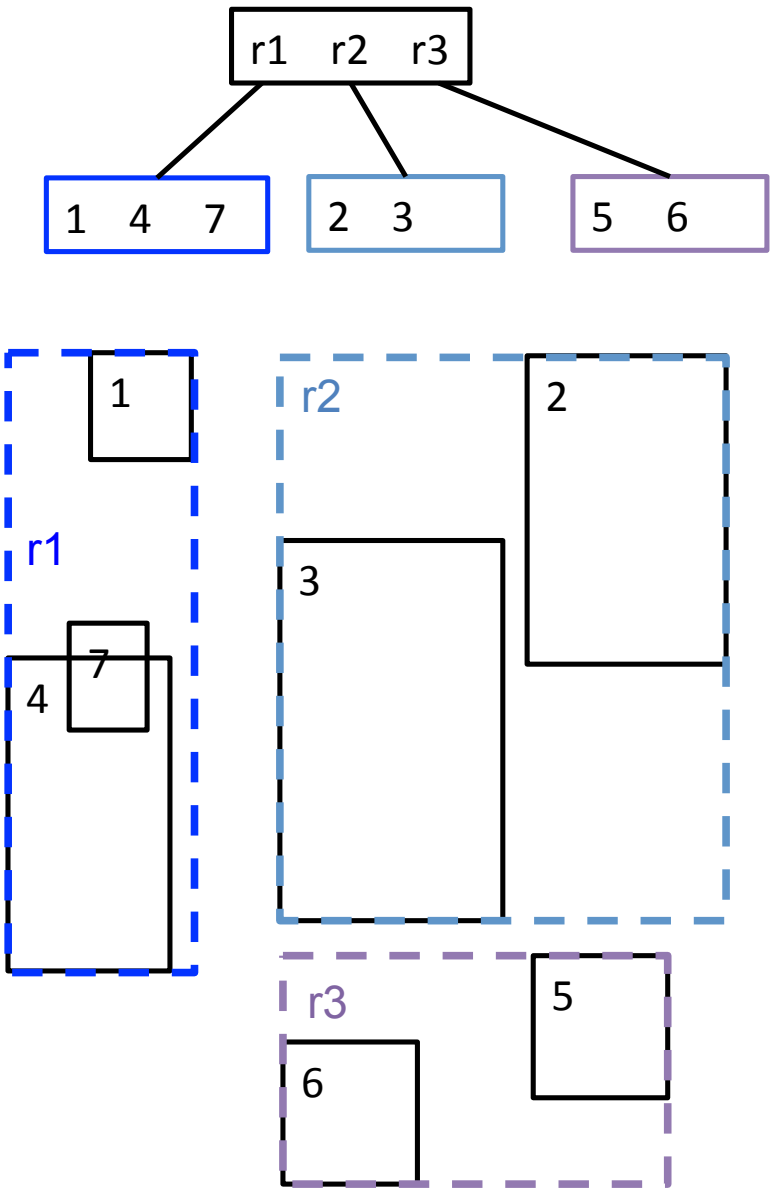
Adding the fourth region requires a split. We must choose which regions to group together. Ideally regions near each other will be grouped together.



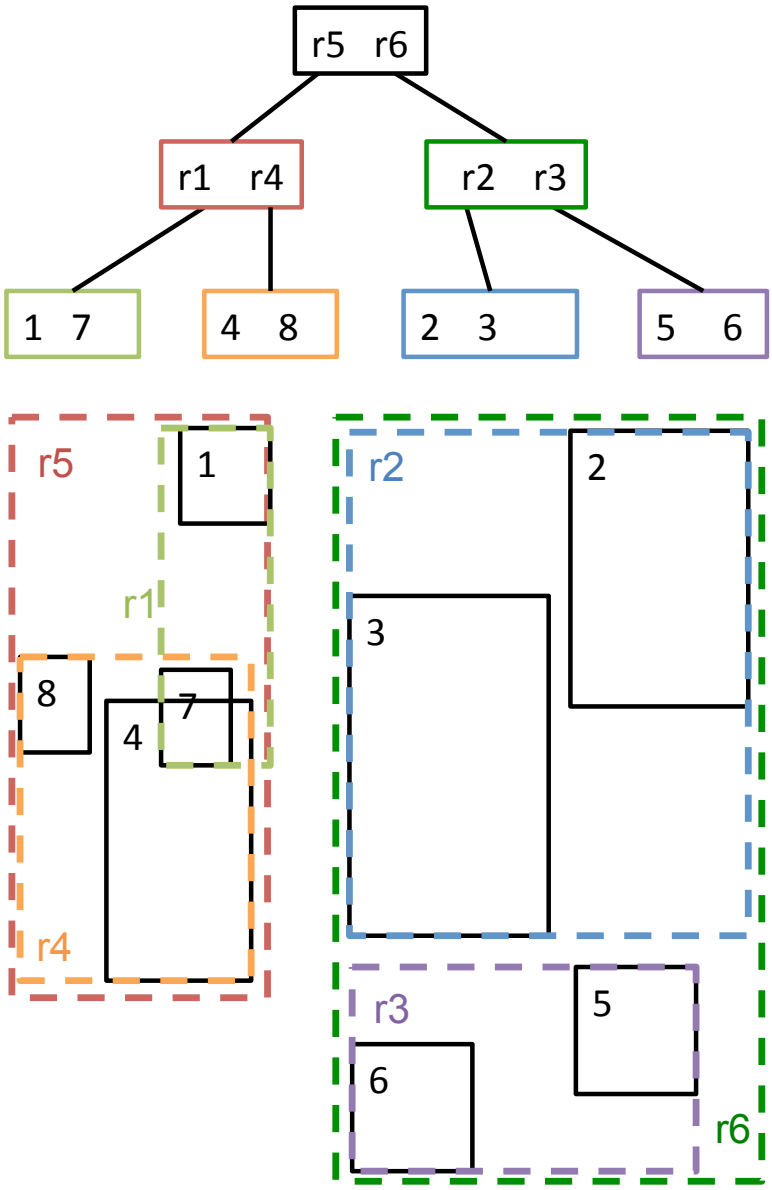
To insert the 5th region, we must choose which bounding rectangle to expand. Again, we want to use some heuristic, such as the expansion that covers the least additional area.



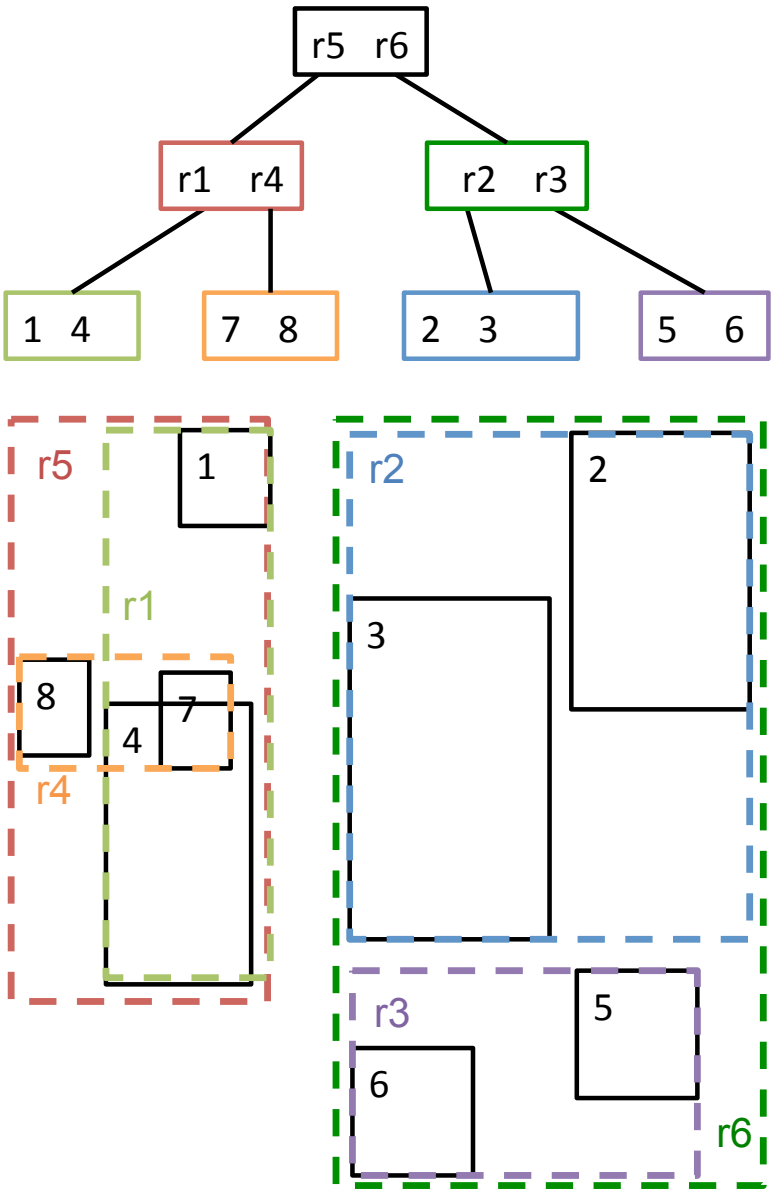
Adding the 6th region requires another split, and the root node is now full. We go ahead and add the 7th region since it does not force a split



Finally, the eight node will force a split, and the promoted bounding rectangle will also force a split, causing a split all the way up to the root



An alternative, and better, choice of bounding rectangles when inserting the 8th region. Why is it better?



Note that algorithms exist to efficiently build an R tree from a set of input regions. Such algortihms can have much better performance than continously inserting regions one by one because they avoid node splitting and tree growing, resulting in fewer disk I/Os.

Queries

What is the procedure to compute a nearest neighbor to region 1? In the how many disk pages do we read? Repeat for Region 7.

What is the procedure to compute a range query: "Find all points in the rectangle defined by an MBR surrounding regions 7,3, and 5? How many disk page reads?

Advantages:

- Adapts to distribution of spatial objects.
- Aligns well with disk architecture

- Handles all types of spatial objects well, and generalizes to higher dimensions.
- Bulk loaders exist.

Disadvantages:

- Can degrade to linear time for operations (why? provide an example).
- Performance can degrade if frequent inserts/deletes made (choices of MBRs).

Various heuristics can be used to choose which spatial objects to group together in a node. They generally involve tradeoffs in proximity/ empty space coverage/ thinness of MBRS.

R-trees are supported in virtually all commercial spatial databases. Lots of approaches have been developed to get around the problem of degenerate R-trees: multiple R-trees designed around the particular application (drones taking hi-resolution images), using the boundary line segments as the objects being stored instead of the entire regions, etc.

Variants: R*tree (different splitting and MBR choosing criteria, R+tree)

CHAPTER

THREE

BIG DATA

This module discusses approaches to handling extremely large data sets, generally in an extremely parallel manner.

3.1 Concepts

Big data is a general term applied to a large variety of situations, and with numerous definitions. In general, big data concepts tend to come into play when one or more of the **three V's** of big data begin to be expressed in a particular problem or applciation. The three V's of big data are:

- Volume: there is a lot of data. Typically things get interesting when the terabyte mark is exceeded, and get more interesting from there. Traditional databases and traditional data techniques can generally handle up to a TB of data without having to get too complex (in terms of guaranteeing uptime, data duplication and backups, and performance).
- Velocity: The data is generated, or arrives at the data center, very quickly. Anything that generates lots of data in a short time. For example, experiments in te large hadron collider create up to PB of data in less than second. Facebook users are updating their accounts all the time.
- Variety: The data is not a fixed format, fixed size, or even consists of a fixed type of data. Traditional databases require the decleration of relations with explicit structure, and usually, fixed size records. This is impossible in mnay big data settings. Perhaps the data being stored evolves over time, perhaps the structure of records evolves over time, perhaps not all records are complete. The system must be able to not only handle such data, but must be able to query data in a variety of formats

Big data systems wrestle with problems of scale that are just not an issue in smaller systems. Systems may comprise thousands of nodes. Even if the nodes are expensive, server grade hardware, node failures are going to be a problem. Think about creating a new social networking app that stores user data. The company requires a lot of storage, to store a lot of data items that must be quickly retrievable, that must provide some guarantee that data will not be lost, and that must do all this on a budget. Thus, central design goals are typically to be:

1. scalable: be able to grow the systems with minimal configuration, and with heterogeneous hardware
2. maintainable in the face of failures: if a node fails, the data stored on that node must already be duplicated or triplicated on other nodes, and the sytem should notice when nodes fail and make sure all data that was stored on that node is sufficiently replicated on other nodes. This must be done automatically
3. use commodity, heterogeneous hardware: nodes need to be flexible, the system should be able to grow without upgrading existing nodes if desired
4. provide fast query and retrieval: the system needs some query mecahnism that is highly parallel, flexible, and realtively easy to use. Furthermore the queries must be able to possible return huge results (more data than can fit on a single node, possibly)

To achieve these goals, big data systems have traditionally been designed as a distributed file system combined with a query mechanism

3.2 Google File System (GFS)

[GFS2003]

GFS was what really got the whole big data thing going on a large commercial scale. The idea was to create a file system that can expand over multiple nodes (thousands) automatically, can automatically incorporate new nodes and identify and de-incorporate faulty nodes, and can handle huge volumes of data.

Remember the google application area. Web pages are crawled and stored on google servers. Those web pages are then processed into a form that is easily queryable using massive parallelism. Crawlers are only so fast **(why?)**, and web pages need only be re-crawled so often **(what is the exception?)**, so the velocity of data coming into the servers is not necessarily huge. But, they are keeping a copy of the internet, so volume is huge! Also, web pages are unstructured text, contain multimedia, etc, so variability is huge.

3.2.1 The stated design goals of GFS are:

- Failures are the norm. With a ton of hard drives, MTTF means something will fail all the time (hd, power supply, memory, OS, human error). Also, machines will be built with commodity parts, and environemntal factors may not be kept as strictly controlled as a supercomputer (cooling is a huge cost in a data center).
- Files are huge. file sizes in the GB range are common. Disk page assumptions need to be revisited because they are so much smaller than the data being accesses.
- Files tend to be appended to, rather than overwritten. Once written, files tend to be read only, and often read sequentially (unlike a B+tree file). Think archival data, intermediate results of computations. file appending is the key focus. Ideas such as enforcing atomicity to support caching data blocks at a client is no longer the focus.
- Fit the file system to the application. Consistency model is vastly relaxed. **(what is a consistency model. In what situations are various consistency models required. Think database transactions, etc). How important is it that a user sees the most up to the second version of a web page in a Google search?**. An *atomic append* operation is introduced so multiple clients can append to a file without burdensome synchronization code.

Note

In this case, a file typically means a chunk of data consisting of multiple objects. Thus, a file could be a bunch of web pages, and their associated media. A file is not a single web page. The goal here is to make the file the unit of interest, provide operations on a per file basis, and manage the distribution, replication, and access to files across the file system. However, files are designed mainly around storing groups of web pages and their associated content/meta data.

3.2.2 Main assumptions:

1. Built from inexpensive, failure-prone hardware.
2. Houses a modest number of large files (optimized for multi GB).
3. Two kinds of reads: large streaming reads, small random reads that are typically ordered to progress sequentially through a file.
4. Large sequential writes. Once written, files are usually static.
5. Support for concurrent appends is key.
6. High bandwidth is more important than low latency. Very few stringent response time requirements.

3.2.3 Architecture

- A single master, multiple chunkservers, and multiple clients.
- Files are divided into fixed size chunks.
- Chunkservers store chunks on local disks. Each chunk is replicated on multiple chunk servers.
- Master maintains all metadata
- Client code communicates with master and chunkservers (master for metadata, direct communication with chunkservers for actual data)
- No caching (eliminates cache coherence issues) at client or chunkservers. Chunkservers do utilize a buffer for faster retrieval, but only for local data. Image from [GFS2003]. Note the minimal role of the master. The clients perform most of the action typically associated with a master node, the master just answers client questions, serves up metadata, bookkeeps, and make decisions, but the clients handle all the heavy lifting.

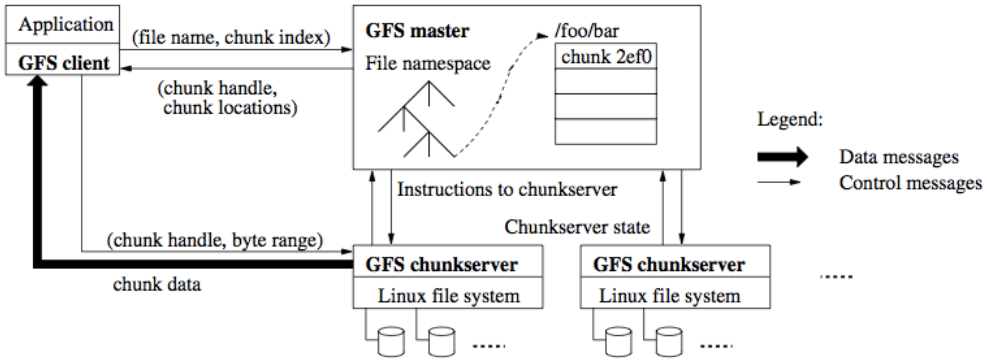


Figure 1: GFS Architecture

Large Chunks vs Small Chunks

A single file can exist with chunks on multiple nodes. Is it better to have a file as a single large chunk, or many distributed smaller chunks. Think in terms of the goals stated above (especially append).

Quote from the paper "However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few

chunkservers storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batch-queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations."

Note the hint about client caching. Again, the management of communication, and the vast majority of querying complexity is handled by the client. This makes the master node only worry about book-keeping, so clients can be upgraded without a major system rewrite!

3.2.4 Metadata

Master maps files to chunks, keeps locations of chunks and replicas. Metadata is stored in memory (a manageable size of files and chunks).

typically, 64 bytes of metadata required for a 64 MB chunk, making in memory metadata limitations feasible.

Master keeps up with chunkservers through **heartbeat** messages. Also used to keep track of which nodes are alive!

Operation Log:

The metadata changes are logged in the operation log. This is critical! it defines the timeline that orders concurrent operations. It is replicated (because it is critical!). Responses to clients are made ONLY after all changes are flushed to local disk of all copies of the log! (remember write ahead logging from databases?) Essentially a redo log. Checkpoints are kept, and logs can be replayed to bring the system up from an interruption.

Consistency:

A file region is **consistent** if all clients see the same data, regardless of which replica is read.

A region is **defined** after a mutation if it is consistent and all clients see the results of the mutation in its entirety. (the complete file write)

Regions can be consistent but undefined if only parts of mutations are seen by all clients. Especially concurrent mutations cause this.

Enforcing defined regions is done by:

1. Applying mutations to chunks in the same order on all replicas
2. Using chunk version numbers to identify stale replicas (maybe a node was down for part of a mutation). Stale replicas are garbage collected.

3.2.5 Performing a write

Quote from [GFS2003]

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (secondary) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.
3. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the net- work topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers

to all the mutations it receives, possibly from multiple clients, which provides the necessary serial- ization. It applies the mutation to its own local state in serial number order.

5. The primary forwards the write request to all sec- ondary replicas. Each secondary replica applies mu- tations in the same serial number order assigned by the primary.
6. The secondaries all reply to the primary indicating that they have completed the operation.
7. The primary replies to the client. Any errors encoun- tered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary repli- cas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the be- ginning of the write.

end quote from [GFS2003]

images from [GFS2003]

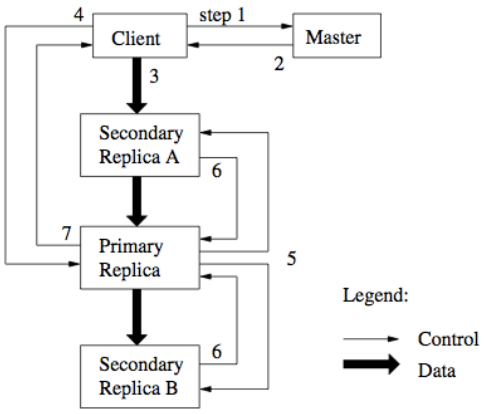


Figure 2: Write Control and Data Flow

3.2.6 Performance

Here is some data from example clusters. Note the amount of metadata vs used disk space. This can be acieved on extremely low end hardware.

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

Lets look at cluster B. For 155 TB of storage, we require 60 MB of data at the master. That is 4 orders of magnitude difference, or roughly .0004 MB per TB. For the chunk servers, there is an order of magnitude difference. Roughly .14 GB per TB. Clearly, the chunk servers need more memory, but the scalability is there. Also, if a chunk server's metadata grows too big, more chunkservers can simply be added to the system to bring down the amount of metadata per node. Also note that the system is reaching capacity at this point. It is 86% full. In practical terms, one would not want to add much more data to the system without increasing capacity.

3.3 MapReduce

[GMR2004]

Once a distribted file system is in place, in this case GFS, we now want to execute some queries! This is where Google's mapreduce framework comes in. As you will see, it fits very nicely around the GFS

3.3.1 Functional Programming, Map, and Reduce

The original concept of mapreduce has its roots in functional programming. Basically, a **map** will apply a function to a set of input data, transform this data in some way, and emit the trnasformed data. A **reduce** will apply a function to a set of input data, and essentially combine that input data in some way so that it returns a smaller set of output data. Lets see an example:

Example

The following python program contains a map function that applies a function to each element of a list that will increment that element. The reduce function then sums the elements.

```
>>> nums=[1,2,3,4,5]
>>> newNums = map( lambda x:x+1, nums )
>>> print newNums
[2, 3, 4, 5, 6]
>>> sum = reduce( lambda x,y: x+y, newNums )
>>> print sum
20
```

Similarly, google mapreduce works by defining a map function and a reduce function. The map function will take objects stored in the GFS, and for each object, willeither emit a key/value pair, or nothing. The reducer then will receive a key k and an array of values that were all mapped to key k. The reducer then performs user defined actions and returns a list of values, again associated with key k. In this way, reducers can be called in parallel.

Recall that one of the advantages of functional languages is that parallelism can be identified rather easily. Similarly, maps can be performed in parallel, and reduces can be performed in parallel as they aggregate final values. In fact, there is no user defined synchronozation, meaning that the computation is as parallel as there are data objects to operate on, and processors. This is known as an **embarrassingly parallel problem**, a bad name for a good problem.

Example

Map and reduce functions have the following signatures:

$map(k1, v1) \rightarrow list(k2, v2)$

$reduce(k2, list(v2)) \rightarrow list(v3)$

So, constucting mapreduce queries is easy for the programmer, just make 2 functions and don't worry about parallelism or synchronization.

Some example applications from the paper:

Example

Count words in bunch of documents quote from paper:

```
map(String key, String value): // key: document name
   // value: document contents for each word w in value:
   EmitIntermediate(w, "1");

reduce(String key, Iterator values): // key: a word
   // values: a list of counts
   int result = 0;
         for each v in values:
            result += ParseInt(v);
         Emit(AsString(result));
```

other examples. quote from paper:

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermedi- ate data to the output.

Count of URL Access Frequency: The map func- tion processes logs of web page requests and outputs [URL, 1]. The reduce function adds together all values for the same URL and emits a [URL, total count] pair. [target, list(source)]

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of [word, f requency] pairs. The map function emits a [hostname, term vector] pair for each input document (where the hostname is extracted from the URL of the document). The re- duce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final [hostname, term vector] pair.

Inverted Index: The map function parses each docu- ment, and emits a sequence of [word, document ID] pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a [word,list(document ID)] pair.Thesetofalloutput pairs forms a simple inverted index. It is easy to augment this com-putation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a [key, record] pair. The reduce function emits all pairs unchanged. This compu- tation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Sec- tion 4.2.

Knowing how GFS works, the architectural diagram of a MapReduce query says it all. From the paper:

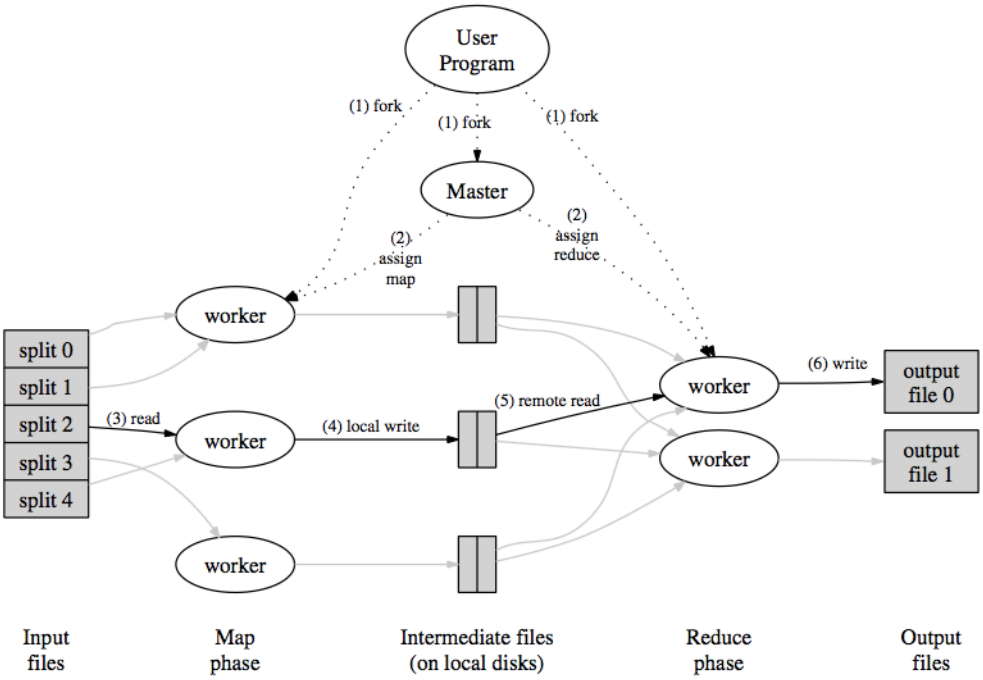


Figure 1: Execution overview

Workers read chunks, apply the maps. The worker gets the chunk information from the GFS master node and accesses the appropriate chunknodes. The results of maps are writted to intermediate files (usually locally, but wherever the master node says there is room). The reducers then start reducing, producing intermediate files along the way. At the end, output files are created, that can be distributed across many nodes! (remember, a file's chunks can be anywhere in the GFS system). So, input data can be huge! (scalable to GFS limitations, which are super high), and output can even be huge!. Also, since the output is also files, the output can be used as input to further MapReduce jobs!

Of course, the system tries to keep data local and take advantage of rack infrastructure. Reducers are first run on a node, then nodes in the same rack are reduced (they share a router), then adjacent racks are reduced (to decrease router trafic), and so on.

Some examples (from the paper):

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

3.4 Bigtable

[GBT2008] So far, we have seen a general distributed file system approach to big data, coupled with a highly parallel mapreduce interface, but some key pieces are missing. First, a file system is very general, there is no notion of data types contained in files, only files, and the querying mechanisms are limited to standard file system operations. Second, mapreduce applies to a specific type of problem, and is not a good fit for every type of query. This is where Google's **Bigtable** comes in. The basic idea of Bigtable is to provide a structured type mechanism for storing items, coupled with a general query mechanism. The general idea is that most of the application Google needed GFS for held data with some sort of structure. For example, a web page record will contain the link (its key), the contents of the page, and links of any web pages that link to that page. This is an extremely basic structure that requires some flexibility (web pages can have different lengths, some web pages will be linked to more than others), but can take advantage of typing, attribute names, and structure to make querying easier.

Bigtable is actually built on top of GFS.

3.4.1 Data Model

The data model is extremely simple... Big table is essentially a map (think data structure map, or hash table). The map is indexed by a (row key, column key, and timestamp) and returns a string. Row keys and column keys are strings, and the timestamp is an integer. Thus, all data is stored as strings, leaving a lot of flexibility. In fact, whatever data you want to store must be encoded into a string!:

Bigtable data map structure

$(row\ key\ (string),\ column\ key\ (string),\ timestamp\ (int)) \rightarrow string)$

Within a particular database, rows are maintained in lexicographic order by row key. Rows are broken into ranges, a single range is stored in a **tablet** (think file from GFS). Thus, the user can control to the locality of data by placing row keys near each other lexicographically. For example, a database of web pages can use URLS as row keys, and reverse their structure such that the TLD comes first, followed by the domain, followed by the subdomain. For example: com.cnn.money, com.cnn.health, etc.

Column keys are grouped into **families**, so that keys of the same family can be compressed and stored together. The intent is for the number of families to be small, but the number of actual colums to be potentially huge. At this point, you should disassociate the term column from any database connotation. In this sense, the family is analogous to a relational data base column, and the Bigtable column is actually a piece of data. Columns are specified by the following: *family:data*, where data is string.

Each cell (column) has a timestamp. Bigtable will keep a user specified number of outdated data versions, the garbage collector will collect any outdated copies that exceed that number.

So, the takeaway here is that Bigtable is meant to store collections such that a collection has a potentially huge number of rows. Each row will have a small number of families, but a potentially huge number of columns. Rows in a collection typically have the same families, but can have different numbers of columns, and each column can be of arbitrary length. So, some structure is implied, but it is up to the user to enforce the structure, not Bigtable. Also, if the user finds the structure must evolve over time, new rows in a collection can simply have different structure. The goal is a balance between structure and flexibility, an easy to use text-based data representation, and the ability to have huge rows, and huge numbers of rows.

Example: storing web pages.

In the paper, Webtable is used as an example. Webtable is a Bigtable database that stores web pages, and the URLs of sites that link to each web page. Each record has the following structure:

```
row key = [reversed url], contents:"<html> ... </html>", anchor:"url of a linking site" t2, ... , anc
```

A timestamp is stored with each column. Multiple versions of a single column can exist (with differing timestamps).

Assume cnn.com is linked to by cs.siue.com and active.com. The resulting record looks like:

$com.cnn \rightarrow$	$contents$: " $< html > ... < /html >$ " t_1	$anchor$: "$cs.siue.com$" t_2	$anchor$: "$active.com$" t_3

Note that the number of anchors is variable depending on the website. Also, each anchor is variable length, and the web page contents are of variable length.

3.4.2 API

An API exists on top of Bigtable, and a mapreduce interface exists. Single row transactions are supported, which allows atomic read-modify-write sequences on a single row. General transactions are not supported.

General transactions were going to be a part of Bigtable, but none of the applications using Bigtable required them. The major source of requests for transactions is to support secondary indexes. Specific functionality to support secondary

indexes is planned. This support will be more efficient than general transactions, but less general.

3.4.3 Implications

The ideas behind Bigtable have found their way into numerous products: MongoDB, Hive, Cassandra, etc. The notion of text based storage, a flexible structure, potentially huge rows (or documents, or whatever the various systems call them), is extremely powerful. Furthermore, because these systems are built on a GFS-type file system, all the advantages of GFS are there: data replication, commodity hardware, easy data replication across geography.

The main drawbacks are lack of enforceable structure of records (like relational databases provide), and lack of transactions (although later work addresses transactions).

If you look at the querying models provided by Bigtable like systems, you notice that they follow the mapping paradigm pretty closely. In general, such systems are referred to as **key,value stores**, since a key is mapped to a value (a row key maps to a row consisting of a bunch of columns). Other systems use the terminology that a key maps to a document. These systems have received much attention and use in a variety of web applications.

Note

What are some good examples of the use of key,value stores in applications?

Think about Facebook... They use a variety of systems to keep track of everything. Where does a key,value store fit in?

What are reasons to not use a Bigtable-like key,value store? Think beyond transactions. (Think about link pointers in B-trees)

3.5 Percolator [Perc2010]

Basic idea: MapReduce and batch processing systems are most efficient with big chunks of data, so a need exists to make small incremental updates. For example, web pages change somewhat frequently, how can those changes be reflected efficiently in GFS? Percolator addresses this issue, allowing the same number of documents to be processed per day as was before, but reducing the average age of documents in Google Search (by about 50%!).

The Problem

Building an index of the web.

1. Crawl every page on the web
2. Process them with respect to some invariants. Example: if a page is crawled under multiple URLS (multiple linking pages), only the one with the highest pagerank gets put in the index. (pagerank must be fixed during this step). Each link is inverted so that the anchor text from outgoing links is attached to the page to which the link points. This must also work across duplicates (links to a duplicate page are linked to the highest pageranked link)

This can be accomplished with mapreduce jobs. One to cluster duplicates, one for link inversion, etc. Note that this style of algorithm construction used natural barrier synchronization. One mapreduce job must finish before the next starts! This makes the invariants trivial to enforce!

What if we want to update the index after a crawl of a small portion of the web. The original way to do it: Re-run the entire process! Its doable, thanks to the scalability of GFS and mapreduce, but that wastes a lot of resources (time, power, etc). Latency for the update is proportional to the size of the repository, not the size of the update!

We need some sort of transactions. Data volume is too big for DBMSs. Bigtable is scalable, but does not allow the enforcement of the invariants listed above.

What is needed is a way to maintain invariants, but allow many, concurrent, small, incremental updates.

The scale of the repository is multiPB.

Term : **snapshot isolation**: All reads within a transaction see the same copy of the database

Term: **ACID** transactions: atomic, consistent, isolated, durable.

Percolator provides ACID snapshot isolation semantics.

3.5.1 Transactional overview

Percolator is built as a client application (remember, master, server client) on top of big table. Recall that bigTable uses tablet servers to retrieve tablets. Tablets are stored in GFS on chunkservers as files. Users can control the grouping of items in a bigTable table. BigTable provides per-row transactional support (by using timestamps). Percolator will provide multi-row tansactional support

Multiple versions of a data item are stored (bigTable's version number). This provides snapshot isolation (reads occur at a given timestamp). Writes appear at a later timestamp.

Require locks. BigTable is used to implenmt locks. Locks are stored in columns for the row that they apply to. (remember, we can always add or remove columns since there is no strict schema).

Requires a timestamp oracle, to get monotonically increasing timestamps throughout the system.

Procedure:

- First phase is *prewrite*. try to lock ALL cells being written to. If another transaction is writing to a cell, ABORT! (snapshot isolation does NOT provide serializability, so we must abort). If any lock is not available, abort. (how does this compare to traditional databases?)
- Second phase . Get a commit timestamp from the oracle. release locks and make written data visible with a write-record value written to the cell.
- Reads that encounter a lock must wait for the lock to be released (data is being written to that cell). (because this is distributed, not all copies are updated at the same instant. A read may have started at a timestamp whose commit is still percolating through the system).
- Clients can fail (percolator clients), leaving bad locks around. Percolator will clean up locks once it is SURE that a client is dead. It cannot do this, so every transaction has a primary lock. A client cannot complete a transaction without owning the primary lock. If percolater thinks a client is dead, it just steals the primary lock.
- Note that all this means that performance will not be ideal, it turns out to be on the order of seconds, not acceptable in a DMBS, but perfectly acceptable here.

3.5.2 Performance

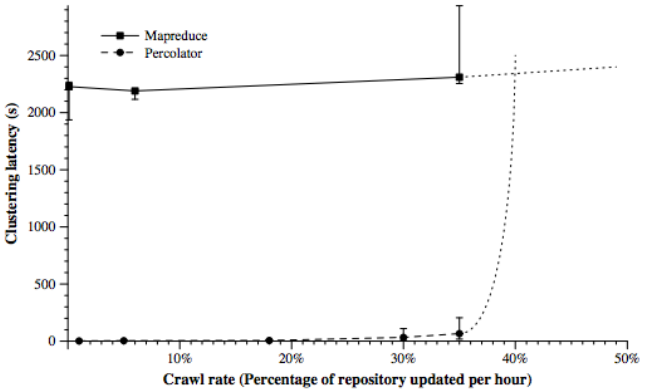


Figure 7: Median document clustering delay for Percolator (dashed line) and MapReduce (solid line). For MapReduce, all documents finish processing at the same time and error bars represent the min, median, and max of three runs of the clustering MapReduce. For Percolator, we are able to measure the delay of individual documents, so the error bars represent the 5th- and 95th-percentile delay on a per-document level.

3.5.3 Implications

So now we have transactions! sort of. How do these transactions compare with traditional DBMS transactions?

How can constraints be implemented in this system?

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

[GFS2003] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 29-43. DOI=10.1145/945445.945450 http://doi.acm.org/10.1145/945445.945450

[GMR2004] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 10-10.

[GBT2008] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI=10.1145/1365815.1365816 http://doi.acm.org/10.1145/1365815.1365816

[Perc2010] Daniel Peng, Frank Dabek, Large-scale Incremental Processing Using Distributed Transactions and Notifications, Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, USENIX (2010).