Outline

1. Access Methods
2. B+-Tree
3. Beyond B+-Trees
4. R-Tree and Variants
5. GiST: Generalized Search Trees
Access Methods: Why?

- Most queries have predicates in them
  - Accessing only the needed records key in performance
Most queries have predicates in them
- Accessing only the needed records key in performance

How relations are stored?
- Heap files: sequential scans, very very fast
- Index structures: random accesses to the needed data
- Scan performance increasing much faster than seeks
  - Must perform *much better* than Scan
  - No point in building indexes on small relations
Access Methods: Why?

- Most queries have predicates in them
  - Accessing only the needed records key in performance

- How relations are stored?
  - Heap files: *sequential* scans, very very fast
  - Index structures: *random* accesses to the needed data
  - Scan performance increasing much faster than seeks
    - Must perform *much better* than Scan
    - No point in building indexes on small relations

- Note the emphasis on “queries”
  - Utility depends more on query workload than data
Most queries have predicates in them
  Accessing only the needed records key in performance
How relations are stored?
  Heap files: **sequential** scans, very very fast
  Index structures: **random** accesses to the needed data
  Scan performance increasing much faster than seeks
    Must perform *much better* than Scan
    No point in building indexes on small relations

Note the emphasis on “queries”
  Utility depends more on query workload than data
Why not use in-memory indexes?
  Data exchange with disks in units of “blocks”
Support iterator interface:
- open (possibly with selection condition)
- get_next, close, insert, delete, update_field
Support iterator interface:
  - open (possibly with selection condition)
  - get_next, close, insert, delete, update_field

Performance goals:
  - Disk I/O (or time) for lookups, inserts, deletes
  - cold vs hot lookups
  - Compare to sequential (seek times improving much slower)
Access Methods

- At a high level:
  - *partition*: partition a dataset or domain into buckets
  - *label*: provide a label for each bucket
  - Sometimes hierarchically (trees), sometimes not (hashing)
Access Methods

- At a high level:
  - *partition*: partition a dataset or domain into buckets
  - *label*: provide a label for each bucket
  - Sometimes hierarchically (trees), sometimes not (hashing)

- Key Differentiating Factors
  - Data (1-d vs 2-d vs n-d, points vs intervals vs spatial objects vs images etc...)
  - Query types (equality, range, nearest-neighbor etc..)
Access Methods

- At a high level:
  - *partition*: partition a dataset or domain into buckets
  - *label*: provide a label for each bucket
  - Sometimes hierarchically (trees), sometimes not (hashing)

- **Key Differentiating Factors**
  - Data (1-d vs 2-d vs n-d, points vs intervals vs spatial objects vs images etc...)
  - Query types (equality, range, nearest-neighbor etc..)
  - Balanced (B+-tre, R-Tree) vs Unbalanced (Quad-tree)
    - Balanced → predictable, uniform performance, but hard to guarantee
    - Typically requires rearranging of labels, splits etc..
Key Differentiating Factors

- Data- vs Space-partitioning
  - Data-partitioning: the buckets are disjoint, but the labels may not be
    - May have to follow down the tree along multiple paths (e.g. R*-tree)
  - Space-partitioning: the labels are disjoint, buckets may not be
Key Differentiating Factors

- Data- vs Space-partitioning
  - Data-partitioning: the buckets are disjoint, but the labels may not be
    - May have to follow down the tree along multiple paths (e.g. R*-tree)
  - Space-partitioning: the labels are disjoint, buckets may not be
    - e.g. Quad-trees, K-D-B trees
    - May have to duplicate pointers to data items in the leaves (e.g. R+-tree)
Access Methods

Key Differentiating Factors

- Data- vs Space-partitioning
  - Data-partitioning: the buckets are disjoint, but the labels may not be
    - May have to follow down the tree along multiple paths (e.g. R*-tree)
  - Space-partitioning: the labels are disjoint, buckets may not be
    - e.g. Quad-trees, K-D-B trees
    - May have to duplicate pointers to data items in the leaves (e.g. R+-tree)

- B+-trees: disjoint buckets and disjoint labels
Imagine:

- The data is already stored on disk in some arbitrary order and you are not allowed to change it.
- How would you best build a hierarchical index structure on top for equality queries?
Imagine:

- The data is already stored on disk in some arbitrary order and you are not allowed to change it.
- How would you best build a hierarchical index structure on top for equality queries?
  - (Homework Question) Use BloomFilters.
Imagine:

The data is already stored on disk in some arbitrary order and you are not allowed to change it.

How would you best build a hierarchical index structure on top for equality queries?

(Homework Question) Use BloomFilters..

No option is going to work well if the data is really arbitrary and you can’t find something to order by.

But an interesting thought exercise:

E.g. you might discover the third byte is different across blocks, but same within a block.

Clustering of data is critical:

Obvious for 1-d data, not so clear otherwise.
Imagine:

- The data is already stored on disk in some arbitrary order and you are not allowed to change it.
- How would you best build a hierarchical index structure on top for equality queries?
  - (Homework Question) Use BloomFilters.
  - No option is going to work well if the data is really arbitrary and you can’t find something to order by.

But an interesting thought exercise:

- E.g. you might discover the third byte is different across blocks, but same within a block.
- Clustering of data is critical.
  - Obvious for 1-d data, not so clear otherwise.

Not academic question: Imagine building an index over a distributed Grid/P2P data.
Implementation Issues:
- Concurrency & recovery
  - Very important issue
  - Intertwined to a very complex degree
  - Can’t build access methods in vacuum for just querying
- Cost estimation
  - Query optimizer needs this information
- Bulk loading
  - Important – have to be done very often
Outline

1. Access Methods
2. B+-Tree
3. Beyond B+-Trees
4. R-Tree and Variants
5. GiST: Generalized Search Trees
B+-Tree

- Balanced, 50% utilization
  - In practice, allow getting lower when doing deletes
  - Inserts are more common, something will get inserted there soon
- \( O(\log_d(n)) \) search, update, delete costs
  - \( d = \) order of the tree (number of keys per page)
B+-Tree

- Balanced, 50% utilization
  - In practice, allow getting lower when doing deletes
  - Inserts are more common, something will get inserted there soon
- $O(\log_d(n))$ search, update, delete costs
  - $d =$ order of the tree (number of keys per page)

Optimizations

- Key compression
- Bulk loading algorithms
- Faster count queries
  - Maintain counts of tuples in the subtrees at the inner nodes
B+-Tree

- Concurrency: not 2PL - too slow
  - Release locks on upper-level nodes as soon as possible
    - Too many queries want to access them
  - Tricky when doing inserts
    - Higher-level pages may have to be split
  - One Solution: Do “preparatory” splits when inserting
  - Much work of engineering nature, few research papers
B+-Tree

- Concurrency: not 2PL - too slow
  - Release locks on upper-level nodes as soon as possible
    - Too many queries want to access them
  - Tricky when doing inserts
    - Higher-level pages may have to be split
  - One Solution: Do “preparatory” splits when inserting
  - Much work of engineering nature, few research papers

- B-Trees?
  - The inner nodes store pointers to data
  - B+-Tree – all pointers to data are at the leaves
  - B+-Trees make many things significantly easier
    - E.g. Can do a “scan” on the leaves for range queries
Outline

1. Access Methods
2. B+-Tree
3. Beyond B+-Trees
4. R-Tree and Variants
5. GiST: Generalized Search Trees
Indexes

- B+-tree: Optimal for one-dimensional data (for range/equality queries)
Indexes

- B+-tree: Optimal for one-dimensional data (for range/equality queries)
- Linear hashing, extensible hashing: Only equality queries

Range queries: \((20 \lt \text{age} \lt 30) \land (10,000 \lt \text{salary})\)

Space-filling curves: Impose a linear order on the multi-dimensional data (limited applicability)

Grid-files, Quad-trees, K-D-B trees etc.

Nearest-Neighbor queries/similarity searches (very common)

Many indexing structures designed, no real consensus

Golden rule: Must beat sequential scans
Indexes

- B+-tree: Optimal for one-dimensional data (for range/equality queries)
- Linear hashing, extensible hashing: Only equality queries
- Multi-dimensional point data
  - Range queries: \((20 < \text{age} < 30) \land (10,000 < \text{salary})\)
  - Space-filling curves: Impose a linear order on the multi-d data (limited applicability)
  - Grid-files, Quad-trees, K-D-B trees etc.
Indexes

- B+-tree: Optimal for one-dimensional data (for range/equality queries)
- Linear hashing, extensible hashing: Only equality queries
- Multi-dimensional point data
  - Range queries: \((20 < age < 30) \land (10,000 < salary)\)
    - Space-filling curves: Impose a linear order on the multi-d data (limited applicability)
    - Grid-files, Quad-trees, K-D-B trees etc.
  - Nearest-Neighbor queries/similarity searches (very common)
    - Many indexing structures designed, no real consensus
    - Golden rule: Must beat sequential scans
Indexes

- Multi-dimensional **spatial** data (*regions, areas etc.*)
  - Queries: find all objects that contain this point, find objects that overlap this object
  - R-Tree and variants
Indexes

- Multi-dimensional **spatial** data (*regions, areas etc.*)
  - Queries: find all objects that contain this point, find objects that overlap this object
  - R-Tree and variants

- Intervals (e.g. time periods associated with events)
  - Queries: Find intervals containing this point, find overlapping intervals etc...
  - Several optimality results exist (see work by Lars Arge, Jeff Vitter et al.)
Indexes

- Multi-dimensional **spatial** data *(regions, areas etc.)*
  - Queries: find all objects that contain this point, find objects that overlap this object
  - R-Tree and variants

- Intervals (e.g. time periods associated with events)
  - Queries: Find intervals containing this point, find overlapping intervals etc...
  - Several optimality results exists (see work by Lars Arge, Jeff Vitter et al.)

- XML ?
  - Some work, but generally considered very hard
Indexes

- Multi-dimensional **spatial** data (*regions, areas etc.*)
  - Queries: find all objects that contain this point, find objects that overlap this object
  - R-Tree and variants

- Intervals (e.g. time periods associated with events)
  - Queries: Find intervals containing this point, find overlapping intervals etc...
  - Several optimality results exists (see work by Lars Arge, Jeff Vitter et al.)

- XML
  - Some work, but generally considered very hard

- GiST: Generalized Search Tree (next class)
Indexes: A Timeline

Multidimensional Access Methods; Gaede, Gunther; ACM Surveys 1998
Indexes

- Much work since then as well
- When reading these papers, ask yourself:
  - Does it beat sequential scan sufficiently?
  - Is the data/workload realistic?
  - Are there other natural workloads on which it may not do well?
- Little rigor in this area
- Some theoretical work, but problems not easy
  - “Curse of Dimensionality”
Figure: R-Tree
R-Tree

- Multi-dimensional, spatial data (points, rectangles)
- Queries: point in polygon, polygon in polygon, overlaps polygon, contains polygon
- labels: bounding rectangles
R-Tree

- Multi-dimensional, spatial data (points, rectangles)
- Queries: point in polygon, polygon in polygon, overlaps polygon, contains polygon
- *labels*: bounding rectangles
- Bulk loading? Hard...
- Search: Follow all paths.
R-Tree

- Multi-dimensional, spatial data (points, rectangles)
- Queries: point in polygon, polygon in polygon, overlaps polygon, contains polygon
- *Labels*: bounding rectangles
- Bulk loading? Hard...
- Search: Follow all paths.
- Insert: Driven by minimizing *area enlargement*
- Split algorithms: exhaustive, quadratic, linear
R-Tree

- Multi-dimensional, spatial data (points, rectangles)
- Queries: point in polygon, polygon in polygon, overlaps polygon, contains polygon
- *labels*: bounding rectangles
- Bulk loading? Hard...
- Search: Follow all paths.
- Insert: Driven by minimizing *area enlargement*
- Split algorithms: exhaustive, quadratic, linear
- Delete: re-insert if too small (why?)
R*-Tree

- R*-Tree: An improvement over R-Tree
- Analysis: four optimization metrics?
  - Minimize area covered by a directory rectangle.
  - Minimize overlap
  - Minimize margin
  - Maximize storage *utilization*
R*-Tree

R*-Tree: An improvement over R-Tree

Analysis: four optimization metrics?

- Minimize area covered by a directory rectangle.
- Minimize overlap
- Minimize margin
- Maximize storage utilization

Conflict with each other

- E.g., minimizing area covered conflicts with maximizing storage utilization.
Changes:

- Insertion algorithm slightly different (minimizes “overlap” at leaf level)
- Aggressive re-insertion (30% entries re-inserted at the same level)
  - Causes headaches with concurrency
- Lots of heuristics...backed by experimental analysis...
- Shown to outperform R-Trees in many experimental studies
R+-Tree

- R+-Tree
  - Space-partitioning version of R*-Tree
  - Forces non-overlapping keys
    - So same data item must be inserted into multiple leaf nodes
  - BUT don’t need to follow all paths down to the leaves
Outline

1. Access Methods
2. B+-Tree
3. Beyond B+-Trees
4. R-Tree and Variants
5. GiST: Generalized Search Trees
Motivation: Extensibility

- New applications: GIS, multimedia (e.g. pictures), CAD, libraries, sequence datasets (Bioinformatics) etc...
- OR systems (next class) allow defining new data types
- What about querying over them?
Motivation: Extensibility

- New applications: GIS, multimedia (e.g. pictures), CAD, libraries, sequence datasets (Bioinformatics) etc...
- OR systems (next class) allow defining new data types
- What about querying over them?

Two proposed solutions:

- Option 1: Design new index structures
- Option 2: Try to use an existing index structure
  - E.g. Can use space-filling curves and B+-Trees to support querying multi-dimensional data
  - Limited applicability (only equality/range queries)
  - What if the app needs new type of query?
Motivation: Extensibility

- New applications: GIS, multimedia (e.g. pictures), CAD, libraries, sequence datasets (Bioinformatics) etc...
- OR systems (next class) allow defining new data types
- What about querying over them?

Two proposed solutions:

- Option 1: Design new index structures
- Option 2: Try to use an existing index structure
  - E.g. Can use space-filling curves and B+-Trees to support querying multi-dimensional data
  - Limited applicability (only equality/range queries)
  - What if the app needs new type of query?

- Postgres paper (next class) had an initial discussion
Figure: From: High-Performance Extensible Indexing; Kornacker; VLDB 1999
GiST

- Generalized Search Tree
  - Allows extending data types as well as queries
  - A single data structure that can handle many different index structures
    - So a single code-base
  - How to use?
    - Register six methods with the database system
    - Start inserting/deleting/querying

Question: Is it always a good idea to use a GiST?

No
Some data and query workloads not amenable to indexing (scan preferred)
Ideas later further developed in Theory of Indexability
Generalized Search Tree

- Allows extending data types as well as queries
- **A single data structure** that can handle many different index structures
  - So a single code-base
- How to use?
  - Register six methods with the database system
  - Start inserting/deleting/querying

- Allows indexing arbitrary types of data

**Question:** *Is it always a good idea to use a GiST?*

- No
- Some data and query workloads not amenable to indexing (scan preferred)
- Ideas later further developed in **Theory of Indexability**
Key insight:

- An index structure partitions the input data hierarchically.

Nodes contain between 2 to \(M\) entries (except root).

Leaf nodes: \((p, \text{ptr})\)
- \(p\): predicate satisfied by the record
- \(\text{ptr}\): pointer to actual record

Non-leaf nodes: \((p, \text{ptr})\)
- \(p\): predicate satisfied by all records in the subtree below.
Key insight:

- An index structure partitions the input data hierarchically
- GiST associates a “predicate” with each subtree, that is true for all data items in the subtree
  - Predicates on a single path from root to a leaf may not agree with each other, but must agree with the leaf
GiST

Key insight:
- An index structure partitions the input data hierarchically
- GiST associates a “predicate” with each subtree, that is true for all data items in the subtree
  - Predicates on a single path from root to a leaf may not agree with each other, but must agree with the leaf
- Nodes contain between 2 to $M$ entries (except root)
Key insight:

- An index structure partitions the input data hierarchically.
- GiST associates a “predicate” with each subtree, that is true for all data items in the subtree.
  - Predicates on a single path from root to a leaf may not agree with each other, but must agree with the leaf.
- Nodes contain between 2 to $M$ entries (except root).
- Leaf nodes: $(p, ptr)$
  - $ptr$: pointer to actual record
  - $p$: predicate satisfied by the record
- Non-leaf nodes: $(p, ptr)$
  - $ptr$: pointer to another node
  - $p$: predicate satisfied by all records in the subtree below.
Need to define 6 functions for a new search tree

- Consistent(E, q): given a $E = (ptr, p)$, might $q$ be satisfied by some tuple in the subtree below $ptr$
  - search/querying (search also done when inserting)
- Union: Find new keys
  - inserts (when add a new $E$ to a page)
GiST

Need to define 6 functions for a new search tree

- **Consistent**($E, q$): given a $E = (ptr, p)$, might $q$ be satisfied by some tuple in the subtree below $ptr$
  - search/querying (search also done when inserting)
- **Union**: Find new keys
  - inserts (when add a new $E$ to a page)
- **Compress, Decompress**: used for compressing the keys
  - Required to implement common optimizations
- **Penalty, PickSplit**: Used for deciding where to insert a new object, and how to split a page if needed

Very similar to R-Tree in many regards
Need to define 6 functions for a new search tree

- **Consistent**($E$, $q$): given a $E = (ptr, p)$, might $q$ be satisfied by some tuple in the subtree below $ptr$
  - search/querying (search also done when inserting)
- **Union**: Find new keys
  - inserts (when add a new $E$ to a page)
- **Compress, Decompress**: used for compressing the keys
  - Required to implement common optimizations
- **Penalty, PickSplit**: Used for deciding where to insert a new object, and how to split a page if needed

Very similar to R-Tree in many regards
Search: Query $q$
- Find all pairs $E = (p, ptr)$ such that $\text{consistent}(E, q)$
- Follow down all the pointers
- Somewhat inefficient, can do better for linear orders

Discussion of how to support R*-Tree illustrates the difficulties simulating an index precisely.

But as with all generalized/extensible approaches, you gain in simplicity what you sacrifice in performance.
Search: Query $q$
- Find all pairs $E = (p, ptr)$ such that $\text{consistent}(E, q)$
- Follow down all the pointers
- Somewhat inefficient, can do better for linear orders

Insert/Delete: Keep the tree balanced
- Use the methods Penalty, PickSplit etc, to decide where to insert/delete, how to rearrange
GiST Algorithms

- **Search: Query** $q$
  - Find all pairs $E = (p, ptr)$ such that $\text{consistent}(E, q)$
  - Follow down all the pointers
  - Somewhat inefficient, can do better for linear orders

- **Insert/Delete: Keep the tree balanced**
  - Use the methods Penalty, PickSplit etc, to decide where to insert/delete, how to rearrange

- **Discussion of how to support R*-Tree illustrates the difficulties simulating an index precisely**
  - But as with all generalized/extensible approaches, you gain in simplicity what you sacrifice in performance
Why an index might perform poorly?

- Predicates at inner nodes not effective $\rightarrow$ traverse down unnecessarily

Reason 1: Too much overlap between the data items (e.g., spatial data)
Reason 2: Key compression not good, i.e., the predicates can't approximate the subtree well (e.g., homework question)

Predicates too large in size in number of bytes

If predicates are allowed to be large, then search will be more efficient (fewer paths travelled)

BUT large predicates $\rightarrow$ tree height increases

Trade-off between key compression and search effectiveness
Why an index might perform poorly?

- Predicates at inner nodes not effective → traverse down unnecessarily
  - Reason 1: Too much overlap between the data items themselves (e.g. spatial data)
  - Reason 2: Key compression not good, i.e., the predicates can’t approximate the subtree well (e.g. homework question)
GiST: Analysis

Why an index might perform poorly?

- Predicates at inner nodes not effective → traverse down unnecessarily
  - Reason 1: Too much overlap between the data items themselves (e.g. spatial data)
  - Reason 2: Key compression not good, ie., the predicates can’t approximate the subtree well (e.g. homework question)

- Predicates too large in size in number of bytes
  - If predicates are allowed to be large, then search will be more efficient (fewer paths travelled)
  - BUT large predicates → tree height increases

Trade-off between key compression and search effectiveness
Why an index might perform poorly?

- Poor storage utilization (too much wasted space)
  - Trade-off between this and above factors
  - Better storage utilization increases key overlap
  - Since we may have to force items together that shouldn’t be
- BUT poor storage utilization → tree height increases
**GiST: Analysis;**

- Why an index might perform poorly?
  - Poor storage utilization (too much wasted space)
    - Trade-off between this and above factors
    - Better storage utilization increases key overlap
    - Since we may have to force items together that shouldn’t be
  - **BUT poor storage utilization → tree height increases**

- Complex trade-offs that can only be answered given a dataset and a query workload
The predicates are Bloom filters of the items in the subtree (as in homework)
  Only supports equality queries

Consistent(E, q): Check if “q” ∈ the Bloom filter

Union: Bit-wise union etc...
The predicates are Bloom filters of the items in the subtree (as in homework)

- Only supports equality queries

Consistent($E$, $q$): Check if “$q$” $\in$ the Bloom filter

Union: Bit-wise union etc...

Why bad?

- If the Bloom Filter size is small (say 10 bits):
  - Too much key overlap
  - All bits in the higher level nodes likely to be set to 1
  - Many predicates will satisfy $\text{Consistent}(E, q)$
GiST: Using Bloom Filters as Predicates

- The predicates are Bloom filters of the items in the subtree (as in homework)
  - Only supports equality queries
- \text{Consistent}(E, q): Check if “q” ∈ the Bloom filter
- \text{Union}: Bit-wise union etc...
- Why bad?
  - If the Bloom Filter size is small (say 10 bits):
    - Too much key overlap
    - All bits in the higher level nodes likely to be set to 1
    - Many predicates will satisfy \text{Consistent}(E, q)
  - If the Bloom Filter size large (say 1000 bits):
    - Number of keys per page too low
    - The height of the tree will be large
- Not sure if anybody has formally analyzed this
Much later work at Berkeley: GiST Project Website

- Indexability theory
- Formalisms for analysis: different types of inefficiencies
GiST: Other issues

- Much later work at Berkeley: GiST Project Website
  - Indexability theory
  - Formalisms for analysis: different types of inefficiencies
- AmDB: A visual debugger and profiler
GiST: Other issues

- Much later work at Berkeley: GiST Project Website
  - Indexability theory
  - Formalisms for analysis: different types of inefficiencies
- AmDB: A visual debugger and profiler
- Concurrency, recovery etc: Not addressed in this paper
  - See High-Performance Extensible Indexing
GiST: How extensible is it?

- Generalizes many ideas, but some limitations
  - Recall the discussion of R*-Trees in the paper
GiST: How extensible is it?

- Generalizes many ideas, but some limitations
  - Recall the discussion of R*-Trees in the paper

- From: Generalizing “Search”…; P. Aoki; ICDE 98

- SS-Tree: Similarity search tree
  - For nearest-neighbor queries
  - Records organized in hierarchical clusters
    - For each cluster: store centroid, bounding sphere radius
GiST: How extensible is it?

- Generalizes many ideas, but some limitations
  - Recall the discussion of R*-Trees in the paper
- From: Generalizing “Search”...; P. Aoki; ICDE 98
- SS-Tree: Similarity search tree
  - For nearest-neighbor queries
  - Records organized in hierarchical clusters
    - For each cluster: store centroid, bounding sphere radius
  - Search: Traverse down the tree looking for the sphere closest to the query point
- Several Issues: e.g. Search is not depth-first
- Need a few modifications (see the paper above)
Figure 1. Similarity search using an SS-tree.
(a) Spatial coverage diagram.
(b) Tree structure diagram.