Machine Learning for Data Management Systems

AI for Index Tuning; Multi-d Indexes

Amol Deshpande
February 16, 2023

Outline

- **Index Tuning using AI**
- Multi-dimensional Indexes – background
- Flood and Tsunami

Background

- Index tuning: Predict best indexes for a given dataset and workload
- AutoAdmin Main Steps
  - For each query in the workload, identify good indexes for that query
  - Combine across all queries to come up with potential "configurations"
  - Use the optimizer to decide which configuration will improve performance the most
    - Key intuition: the optimizer cost models are used during optimization anyway

PROBLEM STATEMENT 1. ***Index tuning***: *Given a workload $\mathcal{W} = \{(Q_i, s_i)\}$, where Q_i is a query and s_i is its associated weight, and a storage budget B, find the set of indexes or the configuration C that fits in B and results in the lowest execution cost $\sum_i s_i \cdot cost(Q_i, C)$ for $\mathcal{W}$, where $cost(Q_i, C)$ is the cost of query Q_i under configuration C.*

PROBLEM STATEMENT 2. ***Continuous index tuning***: *Given the number of iterations K, a workload $\mathcal{W} = \{(Q_i, s_i)\}$, where Q_i is a query and s_i is its associated weight, and a storage budget B, find a sequence of configurations $C^1 \cdots C^K$, where the change in configuration $C^k - C^{k-1}$ fits in B at each iteration k and $\sum_{k=1}^{K} \sum_i s_i \cdot cost(Q_i, C^k)$ results in the lowest execution cost for $\mathcal{W}$.*

Problem

- Regressions: New configuration worse for some queries
  - Reason: Optimizer cost models are not that accurate

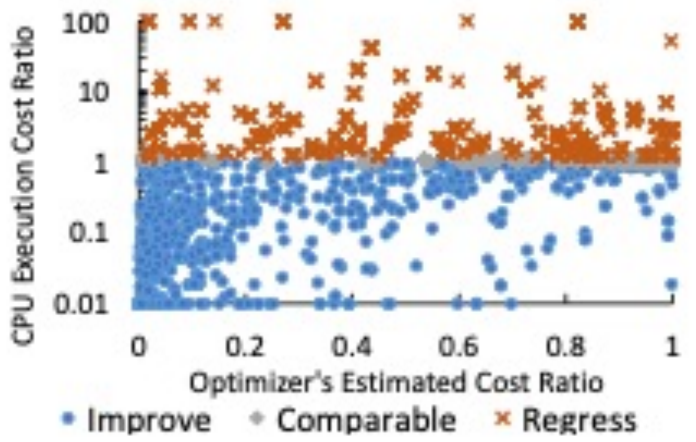


Figure 1: The ratio of CPU execution cost vs. that of optimizer's estimated cost.

Use ML?

- Option 1: Learn to predict the cost of a query plan
  - Too difficult
  - Later work had more success (BAO)
- Option 2:
  - We just need to know if one plan is better than another plan
    - i.e., plans corresponding to same query but different configurations
  - How about we learn a ”classifier”?
    - Should be a much easier problem
- Option 3:
  - Learn to predict the “ratio” of the two costs
    - Slightly easier than Option 1

Architecture

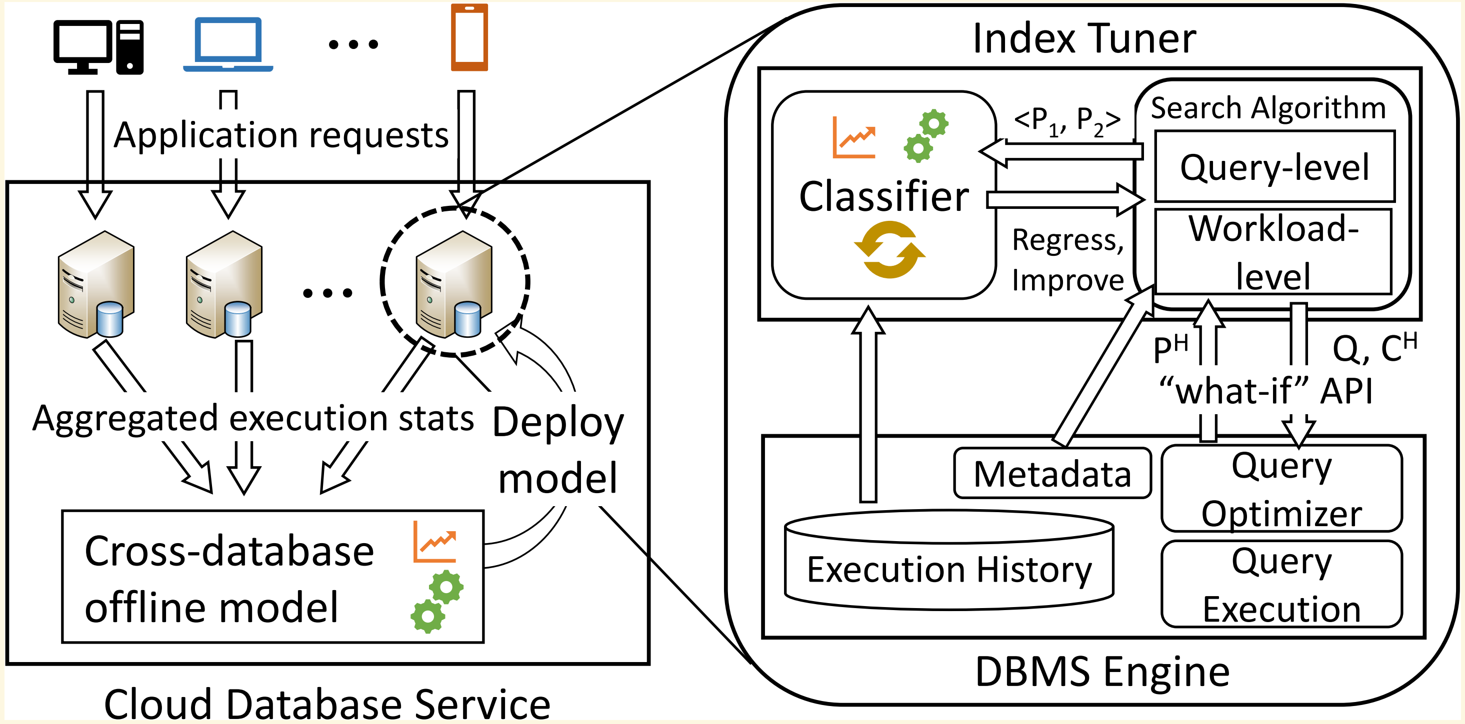


Figure 2: Overview of an architecture leveraging the classifier trained on aggregated execution data from multiple databases in a cloud database service.

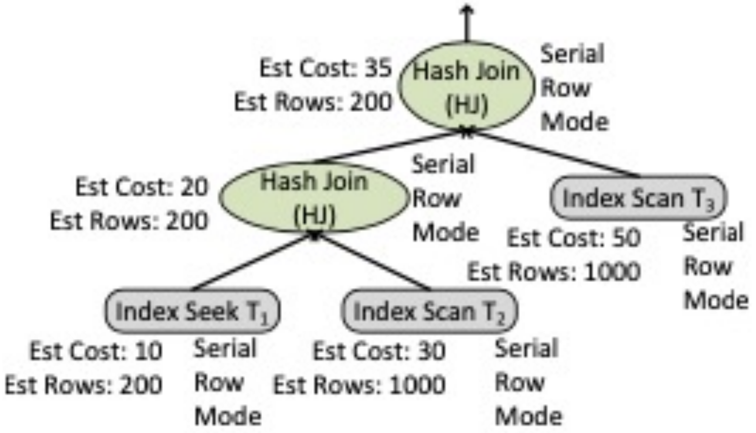
Featurizing Plans

- Use structural as well as local features
- For each operator in the query plan, generate a feature

Table 1: Example feature channels with different ways of weighting nodes encoding different types of information. All estimates are from the query optimizer.

Channel	Description
EstNodeCost	Estimated node cost as node weight (work done).
EstRowsProcessed	Estimated rows processed by a node as its weight (work done).
EstBytesProcessed	Estimated bytes processed by a node as its weight (work done).
EstRows	Estimated rows output by a node as its weight (work done).
EstBytes	Estimated bytes output by a node as its weight (work done).
LeafWeightEst-RowsWeightedSum	Estimated rows as leaf weight and weight sum as node weight (structural information).
LeafWeightEst-BytesWeightedSum	Estimated bytes as leaf weight and weight sum as node weight (structural information).

Featurizing Plans



(a) Example query plan.

EstNodeCost

Seek_Row_Serial	10
Scan_Row_Serial	80
HJ_Row_Serial	55
NLJ_Row_Serial	0
MJ_Row_Serial	0
...	...

...

LeafWeightEstRows WeightedSum

Seek_Row_Serial	200
Scan_Row_Serial	2000
HJ_Row_Serial	4600
NLJ_Row_Serial	0
MJ_Row_Serial	0
...	...

(b) Feature channels for the plan.

Figure 3: An example of encoding a query plan into a vectorized representation called feature channels.

Featurizing Plans

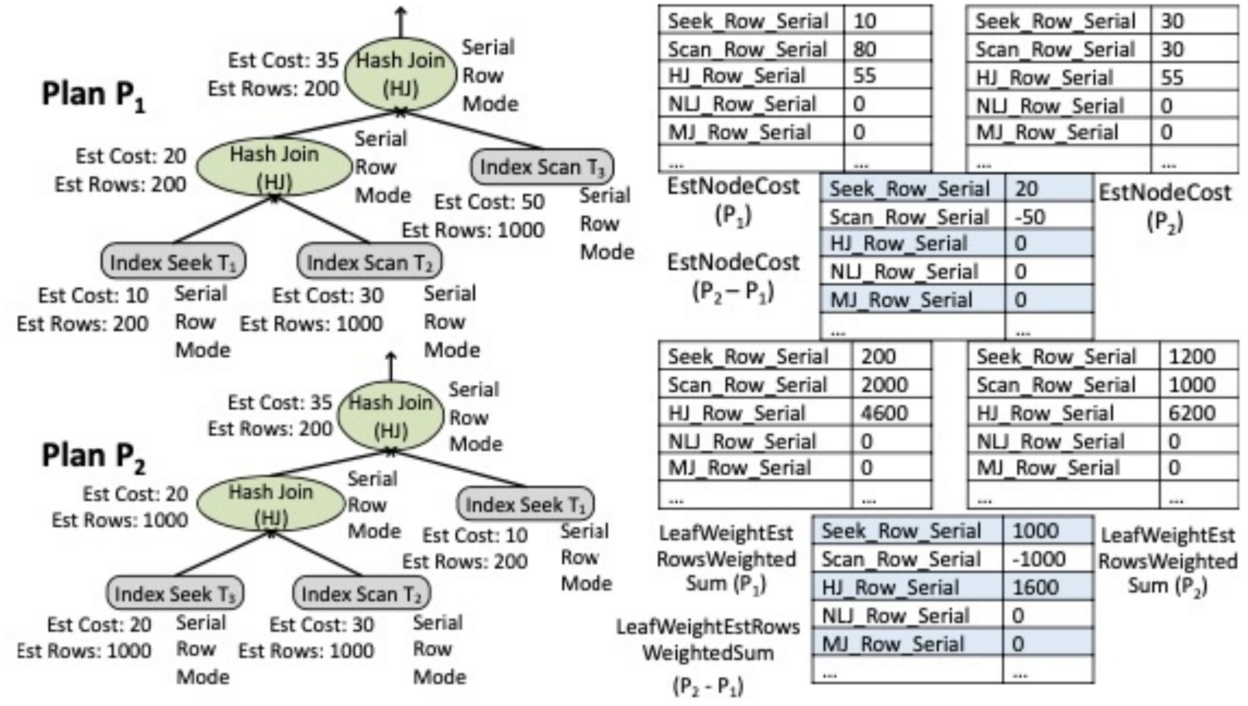


Figure 4: Example of combining the individual plan features into a feature vector for the pair by using a channel-wise difference. Join order change (a structural change) is reflected in the values for channels ending with *WeightedSum*.

Featurizing a Pair of Plans

- Recall: our input to classifier is a pair of plans

$$\frac{ExecCost(\mathcal{P}_2) - ExecCost(\mathcal{P}_1)}{ExecCost(\mathcal{P}_1)} > \alpha$$

- Can concatenate the two feature vectors, but perhaps better to combine them
  - Couple of possible variations

Learning the Classifier

- Can use any off-the-shelf classifier
  - Logistic regression, random forests, gradient-boosted trees, etc.
- Need for adaptation
  - Too many variations from training to real world
- Options:
  1. Learn a model locally for each database -- not enough data
  2. Combine local models and a global model
     - Use local model if the query point is close to training data points (nearest neighbor)
  3. Use the model with less uncertainty about the classification
  4. Learn a “meta” model that tells us which of the two to use

Other Issues

- Integrating with the index tuner
  - Use the classifier to enforce no regression (or limited regression, etc)
  - Still uses the "what-if" API from the earlier paper to get plans for hypothetical configurations
- Other options for learning?
  - Learn to predict the cost of an operator (using similar features)
  - Learn to predict the cost of a plan
    - In either case, use this instead of the optimizer estimate to make decisions
  - Learn to predict the ratio of costs of two plans given the pair feature vector
- Use Deep Neural Networks?

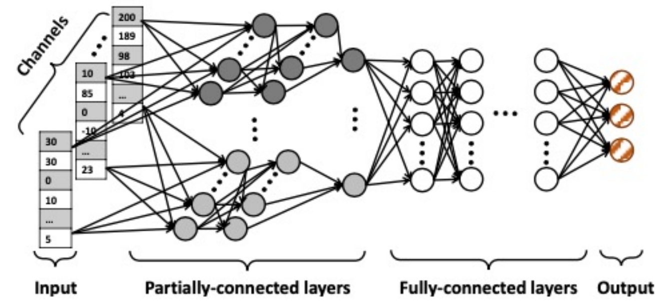


Figure 5: A partially-connected DNN architecture.

Some Results

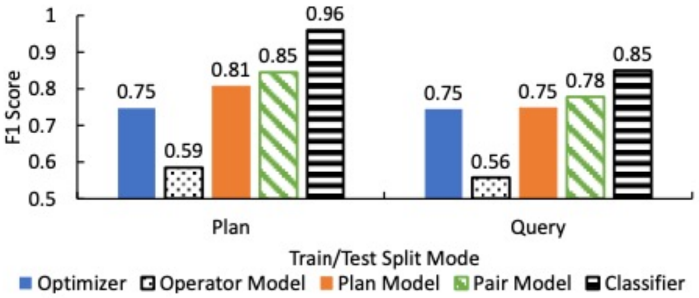


Figure 6: F1 score of different approaches to compare execution costs of a pair of plans.

As is evident, the classifier's F1 score is significantly higher compared to any other model. In particular, *compared with the query optimizer, which is used in state-of-the-art index tuners, for unseen plans, the classifier remarkably increases the F1 score by* 21 *percentage points, equivalent to about* 5× *reduction in the error. For unseen queries, which is a much harder problem to predict, the classifier still improves over the optimizer by* 10 *percentage points, i.e., almost* 2× *reduction in error. Moreover, the classifier is much more accurate compared to any of the regressors.* Interestingly, the operator-level

$$F = \frac{2\mathrm{tp}}{2\mathrm{tp} + \mathrm{fp} + \mathrm{fn}}.$$

Some Results

Table 3: Segmented F1 score for different models, i.e., Optimizer (O), Pair Model (P), and Classifier (C), with the best F1 score for each segment in bold.

Diff Ratio	0.2 – 0.5			0.5 – 1			1 – 2			> 2		
Plan Cost	O	P	C	O	P	C	O	P	C	O	P	C
0-25%				0.70	**0.84**	**0.84**	0.74	0.92	**0.93**	0.85	0.96	**0.97**
25-50%	0.53	0.71	**0.75**	0.63	0.87	**0.89**	0.73	0.92	**0.94**	0.92	0.97	**0.99**
50-75%	0.53	0.77	**0.84**	0.62	0.90	**0.93**	0.71	0.95	**0.97**	0.92	0.98	**0.99**
75-100%	0.50	0.70	**0.81**	0.57	0.86	**0.89**	0.67	0.93	**0.94**	0.92	0.96	**0.99**

pair model = "plan pair regressor" from section 6.1?

Some Results

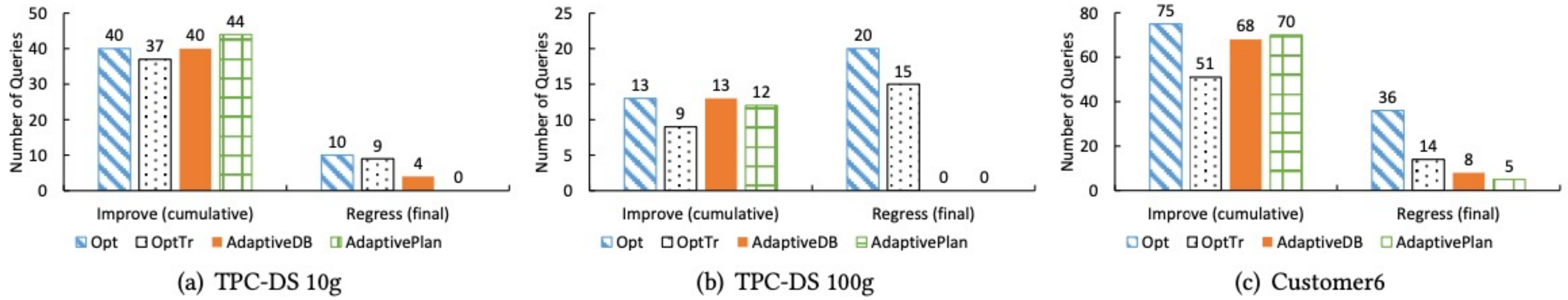


Figure 11: Number of queries improved at its final configuration (with regressed configuration reverted) and regressed at the last iteration for query-level tuning with ten iterations.

Some Discussion Points

- What’s the main take-away from this paper?
- Major concerns with the paper?
- Possible improvements?

Outline

- Index Tuning using AI
- **Multi-dimensional Indexes – background**
- Flood and Tsunami

Different Goals

- Queries on relations with multiple predicates:
  - 10 < R.A < 20 and 20 < R.B < 30
  - Can be done using two separate indexes, but far from optimal
  - Can sort by R.A first, and then by R.B
    - Can't support queries on B alone
- Spatial data
  - Data is points, and queries are rectangles
  - Data is rectangles and queries are rectangles, etc.
- Also different types of queries
  - E.g., find "nearest neighbors" to a given point

Grid Files

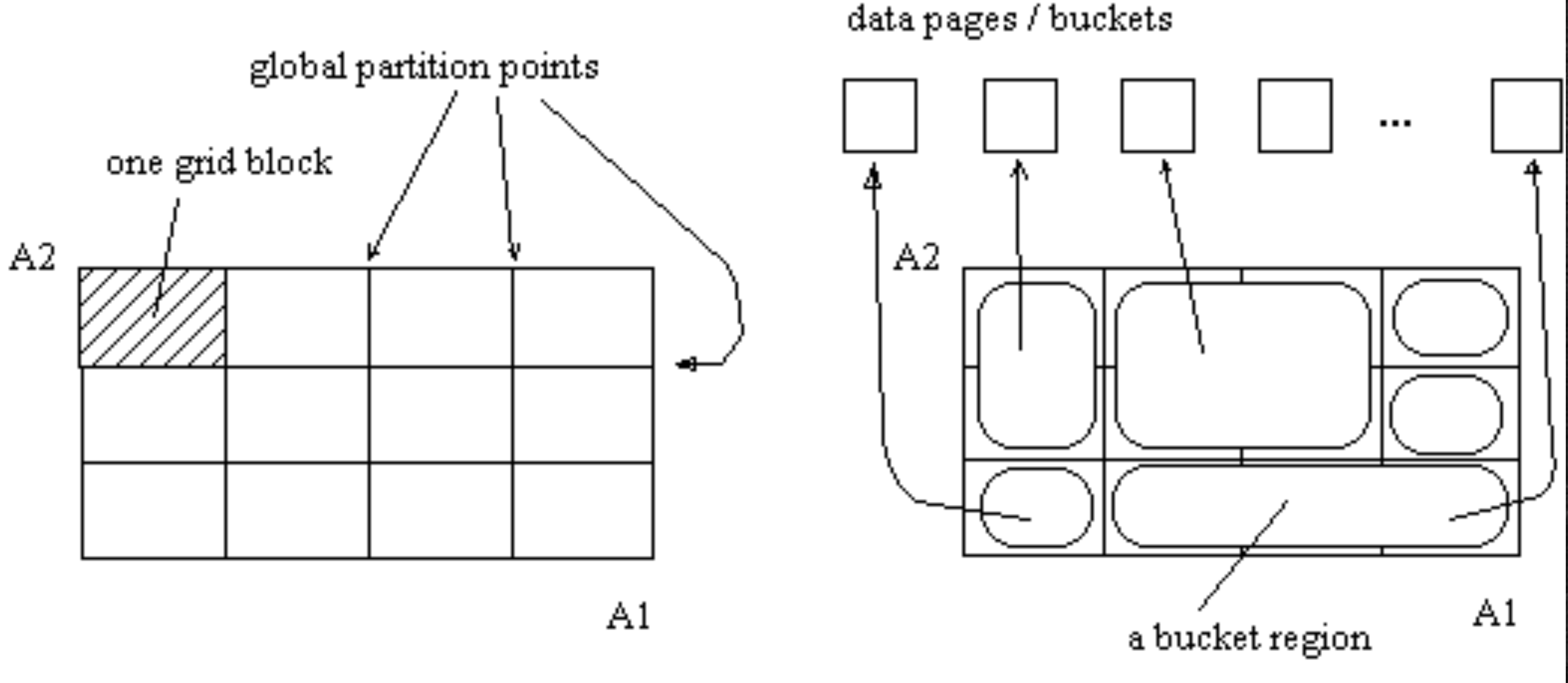
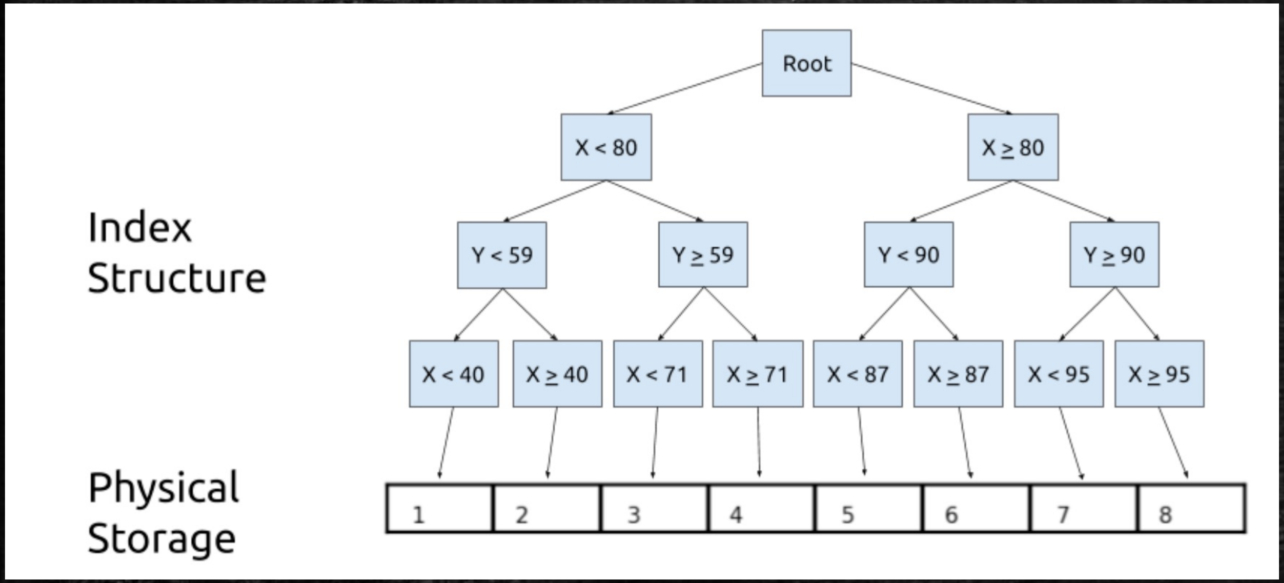


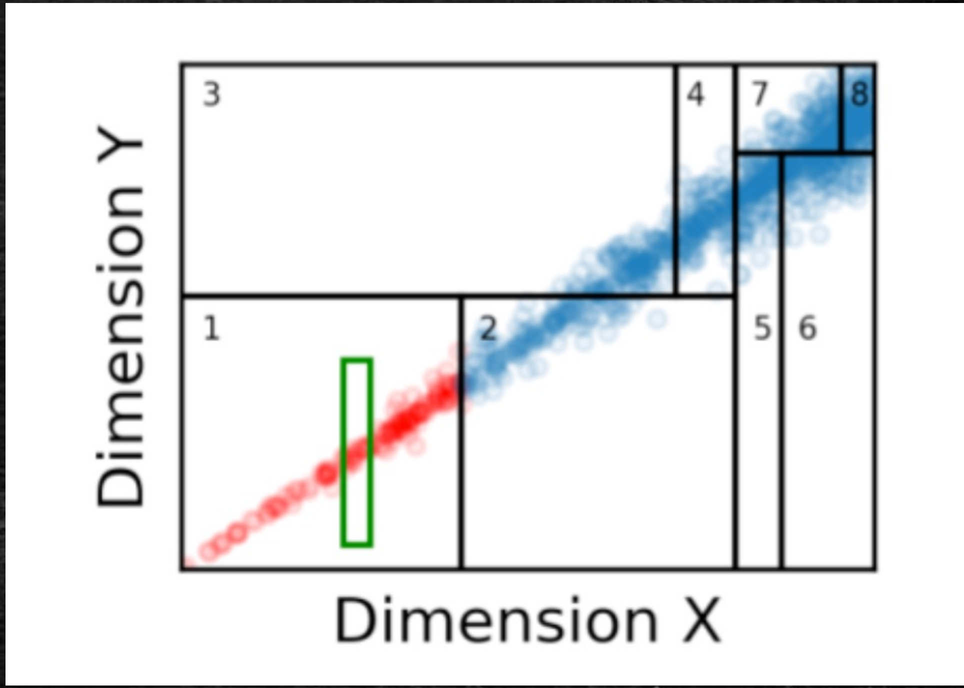
Fig 2.7(a) The Grid File

Fig 2.7(b) Grid block assignment

https://pages.cs.wisc.edu/~zuyu/summaries/db-qual/gridFile

K-d-Trees


Root
X < 80
X ≥ 80
Index Structure
Y < 59
Y ≥ 59
Y < 90
Y ≥ 90
X < 40
X ≥ 40
X < 71
X ≥ 71
X < 87
X ≥ 87
X < 95
X ≥ 95
Physical Storage
1
2
3
4
5
6
7
8


Dimension Y
Dimension X
3
4
7
8
1
2
5
6

R-Trees

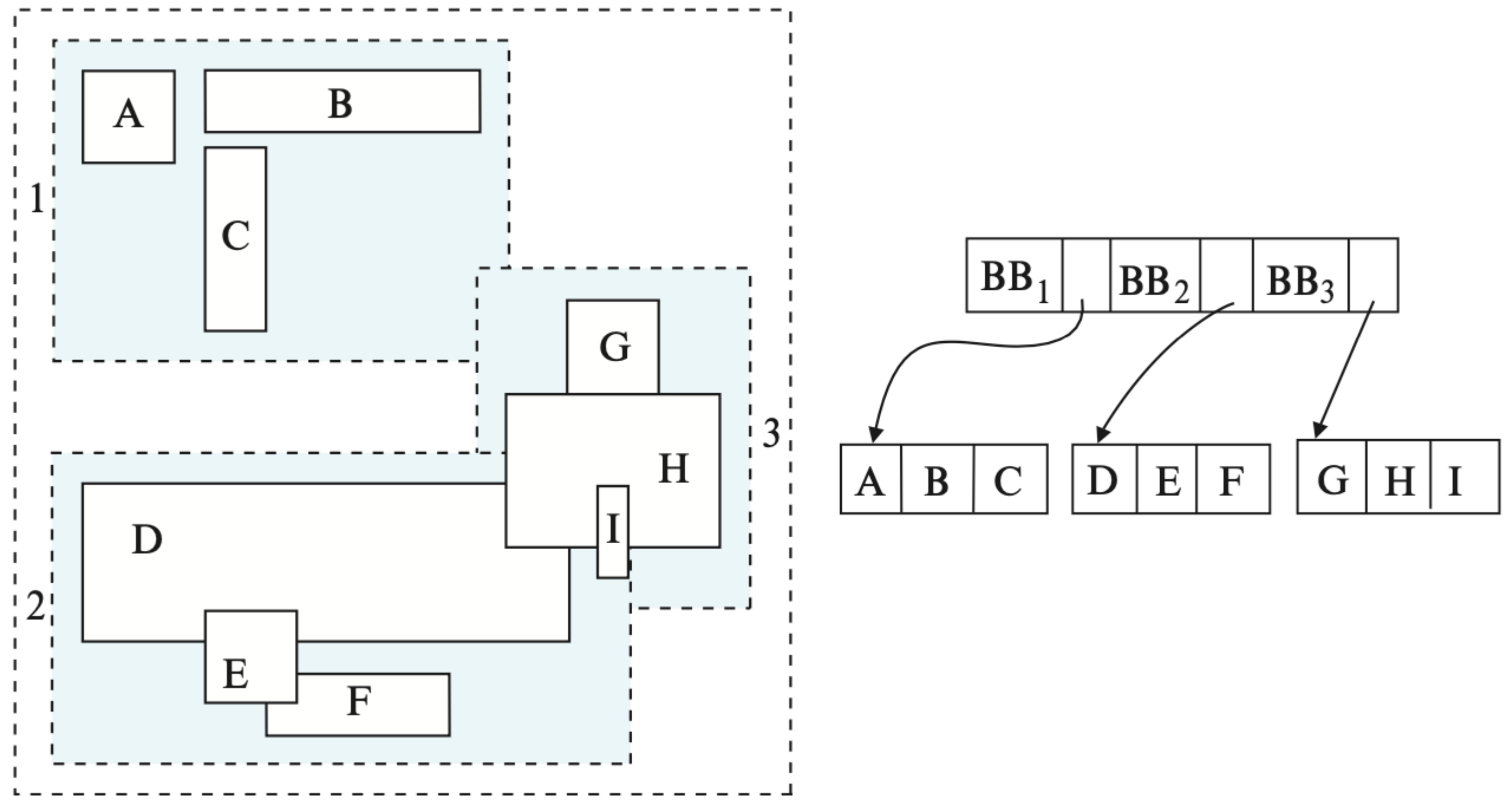


Figure 14.30 An R-tree.

Summary

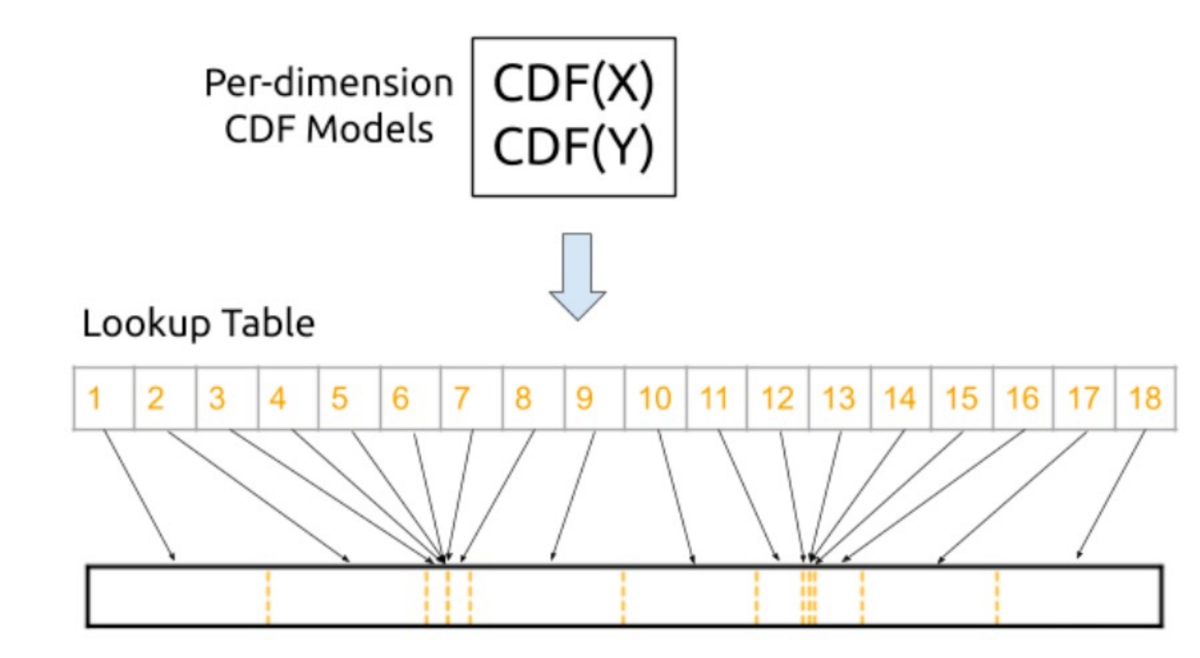
- Work pretty well in small number of dimensions
- Curse of dimensionality
  - Unintuitive behavior in larger dimensions
- Require tuning to work well
- Usually hard to update
  - Most don't support transactions efficiently

Outline

- Index Tuning using AI
- Multi-dimensional Indexes – background
- **Flood and Tsunami**

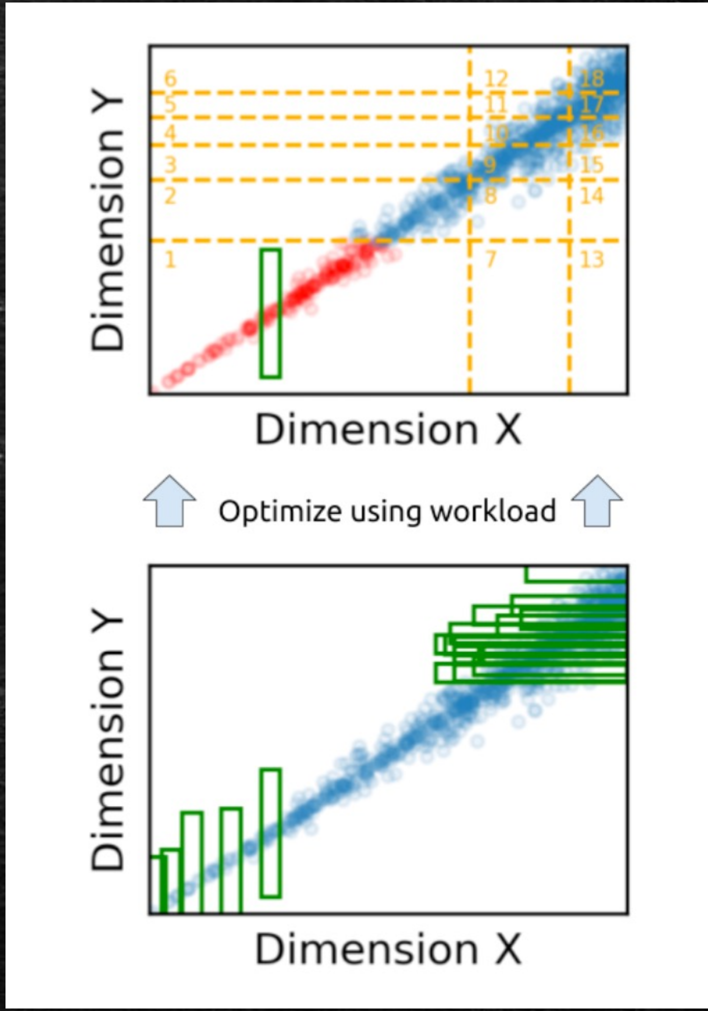
Flood

- For each dimension, figure out an even partitioning (separately)
  - Say 3 partitions for X, and 6 partitions for Y
- For every combination of partitions, add an entry in the lookup table to point to the right block
  - Very similar to Grid Files
- Query for X = 5 and Y = 10
  - First find the partition for X, say 1
  - Then for Y: say 3
  - Then the pointer to the block is in location (x-1)*6 + y = 3



Flood Benefits

- Workload-aware
  - Number of partitions for each dimensions dictated by the overall workload
- Efficiency
  - The CDFs are much more space- and time-efficient than a tree structured index
- Using 50x smaller index size, outperformed traditional indexes by three orders of magnitude



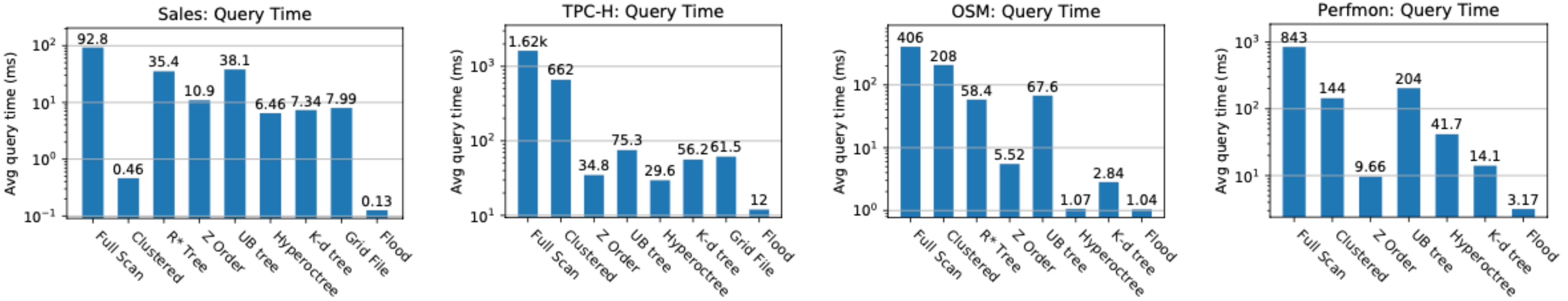


Figure 7: Query latency of Flood on all datasets. Flood's index is trained automatically, while other indexes are manually tuned for optimal performance on each workload. We exclude the R*-tree when it ran out of memory. Note the log scale.

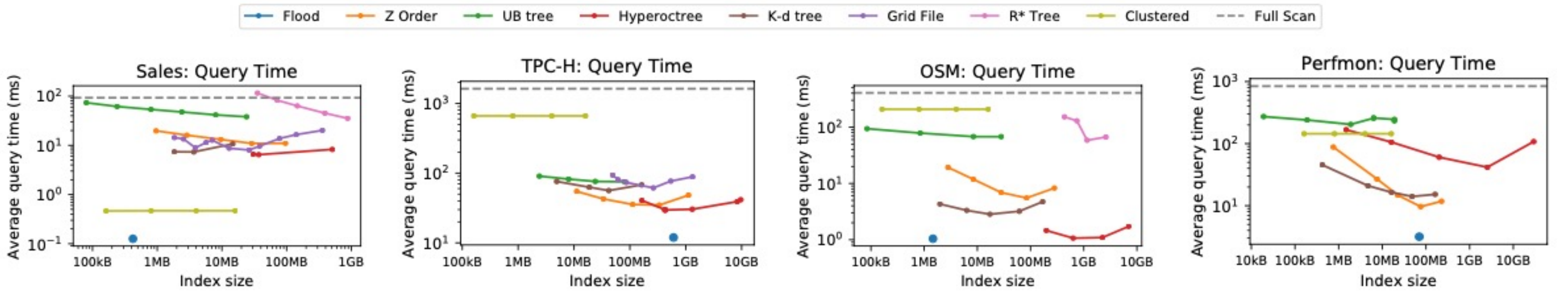


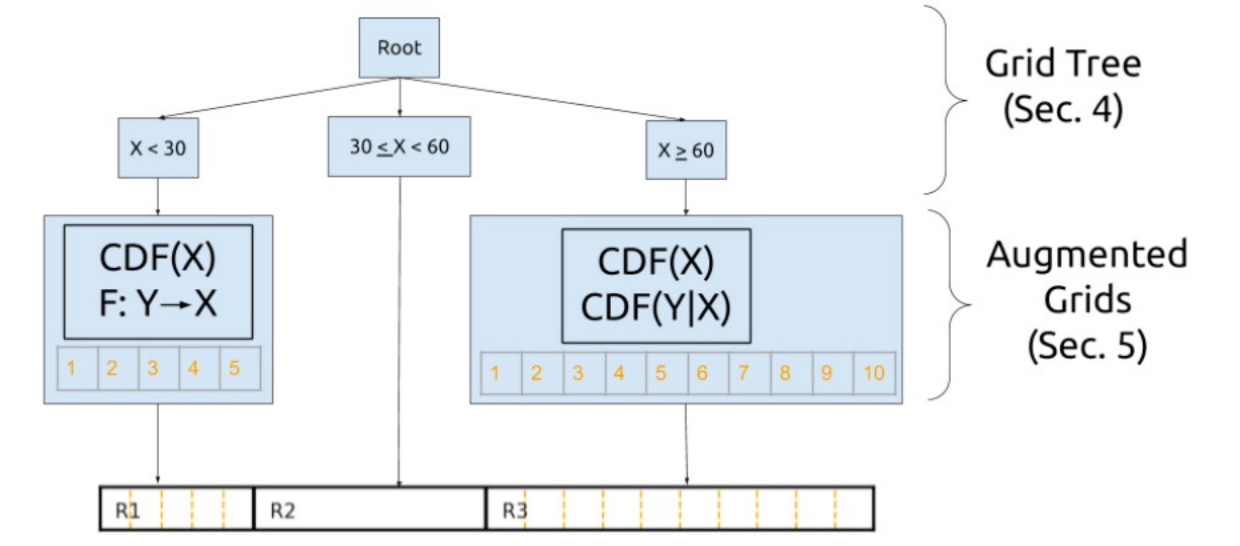
Figure 8: Flood (blue) sees faster performance with a smaller index, pushing the pareto frontier. Note the log scale.

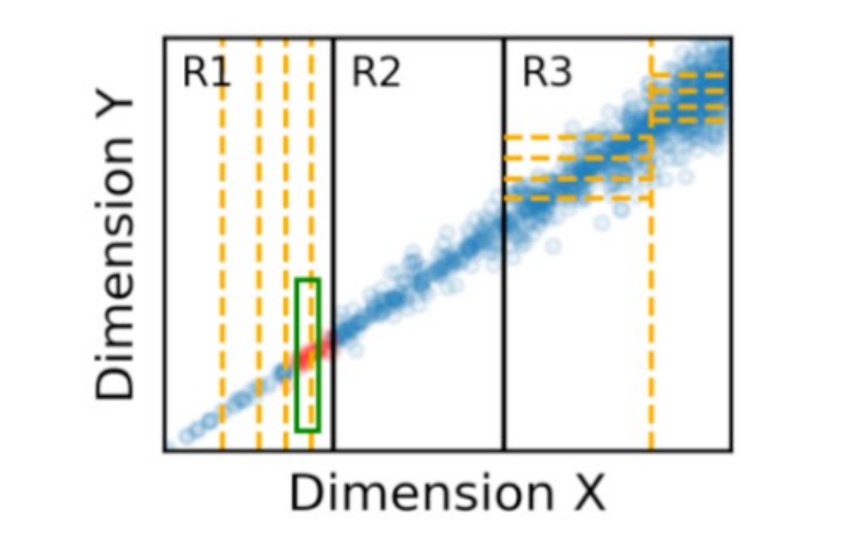
Flood Limitations

- Good average-case performance, but some queries could require scanning large amounts data to extract small results
- Doesn’t handle correlated data well
  - Most data tends to be pretty correlated across dimensions
  - Flood guarantees equal partitions along each dimension, but not across combinations

How Tsunami Fixes This

- Do a coarse-grained partitioning first
- And then, allocate additional resources to each partition as needed





Dealing with Skew

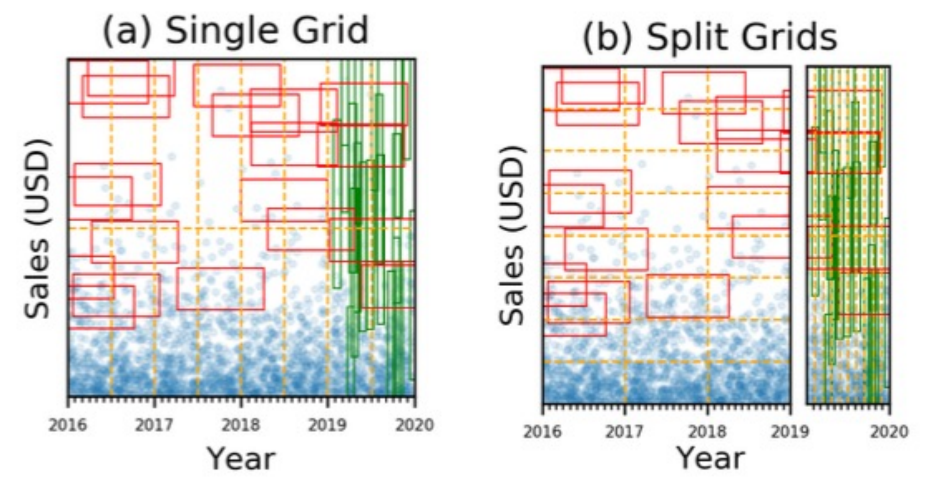


Figure 2: A single grid cannot efficiently index a skewed query workload, but a combination of non-overlapping grids can. We use this workload as a running example.

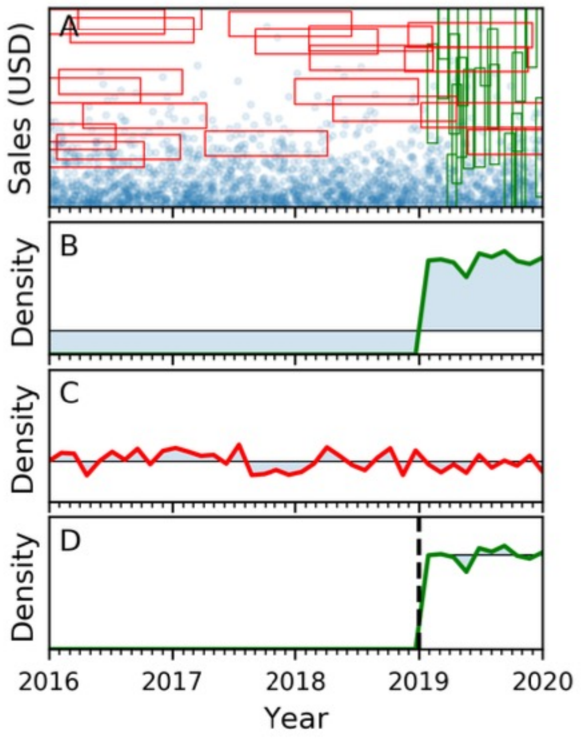


Figure 3: Query skew is computed independently for each query type (Q_g and Q_r) and is defined as the statistical distance between the empirical PDF of the queries and the uniform distribution.

Grid Tree

- Built greedily
- Starting with..
  - root = entire dataspace and entire workload
- Make the "split" decision that most reduces the "skew" along one of the dimensions
  - Skew defined to be the distance between the distribution of queries and uniform distribution along that dimension

Dealing with Correlations

- Key problem: too much variation across the cells
  - Even if each dimension is split evenly
- If very strong monotonic correlation (X almost predicts Y)…
  - Convert the predicate on Y into a predicate on X, and only build an index on X

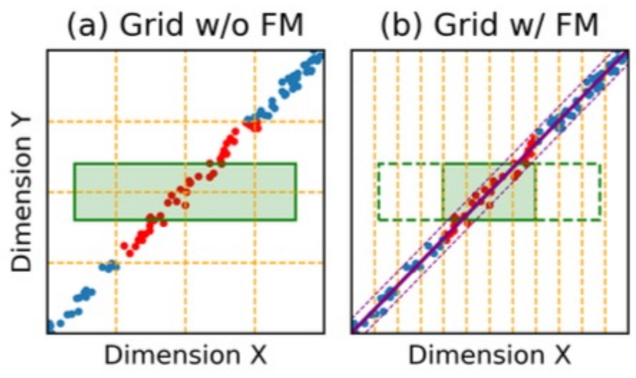


Figure 5: Functional mapping creates equally-sized cells and reduces scanned points for tight monotonic correlations. The query is in green, scanned points are red, and the mapping function is purple, with error bounds drawn as dashed lines.

Dealing with Correlations

- Otherwise use a k-d-tree-like structure to create even cells
  - Except use learned functions instead of a decision tree
- Queries over Y alone are more expensive

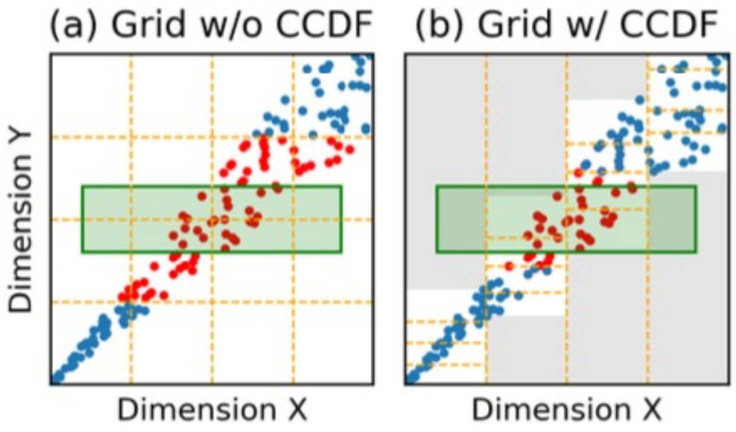


Figure 6: Conditional CDFs create equally-sized cells and reduce scanned points for generic correlations. The query is in green, and scanned points are in red.

Dealing with Correlations

- Some possibilities (“skeletons”)
  - [X, Y→ X, Z] means that we partition on X and Z evenly, and convert any predicate on Y into a predicate on X
- Use an adaptive descent algorithm to greedily find a good skeleton and partitioning
  - Very large search space

Ex. skeleton	$[X, Y \mid X, Z]$ (i.e., $CDF(X)$, $CDF(Y \mid X)$, and $CDF(Z)$)		
One hop away	$[X, Y, Z]$	$[X, Y \mid Z, Z]$	$[X, Y \rightarrow X, Z]$
	$[X, Y \rightarrow Z, Z]$	$[X, Y \mid X, Z \mid X]$	$[X, Y \mid X, Z \rightarrow X]$

Table 2: Example skeleton over dimensions X, Y, Z, and all skeletons one “hop” away. Restrictions are explained in §5.2.1 and §5.2.2 (e.g., $[X \rightarrow Z, Y \mid X, Z]$ is not allowed).

Results

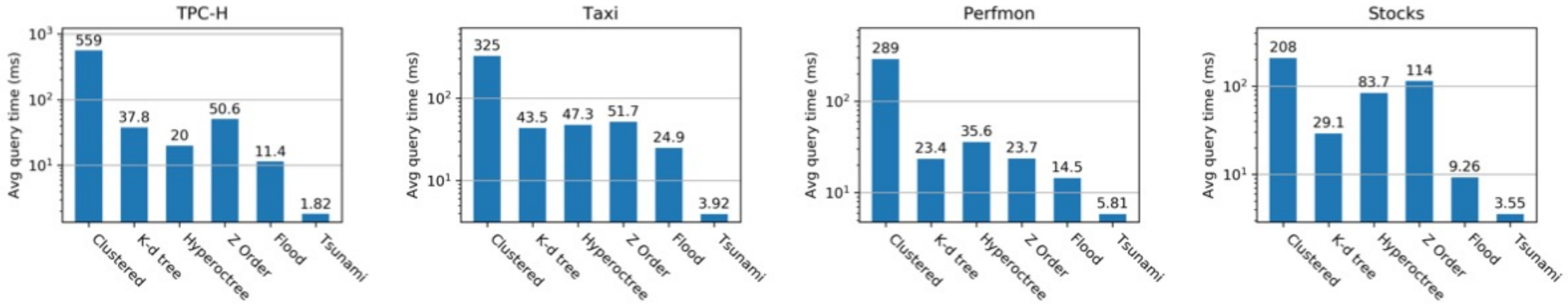


Figure 7: Tsunami achieves up to 6× faster queries than Flood and up to 11× faster queries than the fastest non-learned index.

Some Discussion Points

- What’s the main take-away from this paper?
- Major concerns with the paper?
- Possible improvements?