Advanced Data Management (CSCI 680/490)

Machine Learning in Databases

Dr. David Koop
Checking Computational Results in Systems

Figure 4: Process by which the study was performed.

[Collberg and Proebsting, 2015]
Figure 11: Study result. Blue numbers represent papers that were excluded from consideration, green numbers papers that are weakly repeatable, red numbers papers that are non-weakly repeatable, and orange numbers represent papers that were excluded (due to our restriction of sending at most one email to each author).

[Collberg and Proebsting, 2015]
Excuses for not sharing

- Versioning
- Available Soon
- No Intention to Share
- Personnel Issues
- Lost Code
- Academic Tradeoffs
- Industrial Lab Tradeoffs
- Obsolete HW/SW
- Controlled Usage
- Privacy/Security
- Design Issues

[Collberg and Proebsting, 2015]
Examining 'Reproducibility in Computer Science'

- Repeat the experiment in reproducibility!
- Differences from original
- Shows issues with trying to classify experiments

---

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purported Not Building; Disputed; Not Checked</td>
<td>6%</td>
</tr>
<tr>
<td>Purported Building; Disputed; Not Checked</td>
<td>2%</td>
</tr>
<tr>
<td>Conflicting Checks!</td>
<td>0%</td>
</tr>
<tr>
<td>Misclassified</td>
<td>1%</td>
</tr>
<tr>
<td>Purported Not Building But Found Building</td>
<td>14%</td>
</tr>
<tr>
<td>Purported Building But Found Not Building</td>
<td>0%</td>
</tr>
<tr>
<td>Purported Not Building; Confirmed</td>
<td>0%</td>
</tr>
<tr>
<td>Purported Building; Confirmed</td>
<td>0%</td>
</tr>
<tr>
<td>All Others Purported Not Building</td>
<td>27%</td>
</tr>
</tbody>
</table>

[S. Krishnamurthi et al.]
Reproducible Research

- Science is verified by replicating work independently
- Replication Issues:
  - Requires many resources to replicate (Sloan Digital Sky Survey)
  - Requires significant computing power (Climate Model Simulation)
  - Requires too much time or very specific circumstances (Environment Epidemiology)
- Reproducibility
  - Replication of the analysis based on the collected data (not replicating the data collection itself)
  - Better if we have the actual code or available executables
Reproducibility Spectrum

- **Publication only**
  - Code
- **Publication +**
  - Code and data
  - Linked and executable code and data
- **Full replication**
- **Not reproducible**
- **Gold standard**

[R. D. Peng]
10 Rules for Reproducible Computational Research

• Rule 1: For Every Result, Keep Track of How It Was Produced
• Rule 2: Avoid Manual Data Manipulation Steps
• Rule 3: Archive the Exact Versions of All External Programs Used
• Rule 4: Version Control All Custom Scripts
• Rule 5: Record All Intermediate Results, When Possible in Standardized Formats

[Sandve et al., 2013]
10 Rules for Reproducible Computational Research

• Rule 6: For Analyses That Include Randomness, Note Underlying Random Seeds
• Rule 7: Always Store Raw Data behind Plots
• Rule 8: Generate Hierarchical Analysis Output, Allowing Layers of Increasing Detail to Be Inspected
• Rule 9: Connect Textual Statements to Underlying Results
• Rule 10: Provide Public Access to Scripts, Runs, and Results
(Database) Reproducibility Research Topics

- Design and Management of Experiment Repositories
- Querying and Searching Experiments
- Mining Experiments
Notebook Reproducibility

- Use notebooks from Github (~1 million)
  - Unambiguous cell order? 81.99%
- Study notebook dependencies
  - Dependencies Available? 13.72%
  - Dependencies Install? 5.03%
- Study notebook executability
  - Execute: 24.11% of unambiguous cell order
  - Matched results: 4.03%

[Pimentel et al., 2019]
import pandas as pd

df = pd.read_csv('guardian-top100-female-2019.csv')

In [full]:

df = df.rename(columns={'Age on 1 Dec 2019': 'Age'})

In [over30]:

df = df[df.Age >= 31]

In [under25]:

df = df[df.Age <= 24]
Final Exam

• Monday, May 9, 4:00-5:50pm, PM 153
• Similar format
• More comprehensive (questions from topics covered in Test 1 & 2)
• Will also have questions from graph/spatial/temporal data, provenance, reproducibility, machine learning
Improving Databases
LEARNED AND SELF-DESIGNING DATA STRUCTURES

Stratos Idreos & Tim Kraska
Algorithms rely on the order of data.

Data structures are at the core of any data driven algorithm. In fact for any given problem, the design of the data structure defines the range of algorithms that may be applied.

Algorithms rely on the order of data.

[S. Idreos, 2019]
Data systems rely on algorithms

[Data systems are a collection of many data structures and algorithms.]

[S. Idreos, 2019]
Data structures define performance

As time goes by, data structures become ever more critical for data driven applications.

Jim Gray, Turing Award 1998

register = this room
caches = this city
memory = nearby city
disk = Pluto
Database Questions

How do I make my **data system** run x times as fast? (sql,nosql,bigdata, …)

How do I minimize my **bill** in the **cloud**?

How do I extend the **lifetime** of my hardware?

How to accelerate **statistics** computation for data science/ML?

How do I train my **neural network** x times faster?

---

[S. Idreos, 2019]
Tradeoffs in each structure

no perfect structure

Every data structure design is simply a point in the design space of possible solutions. There is no perfect design. Every design balances the fundamental tradeoffs of Read, Update, and Memory amplification. For example, Read amplification is defined as the excess data an algorithm needs to read on top of the data it wants to read. Typically a data structure would have some kind of metadata or navigation data that help locate the actual data, e.g., the internal nodes of a B-tree. Reading this navigation data is an excess cost, adding to read amplification. Creating a data structure without any navigation data would suffer update or even more read amplification. For example, we could choose to not have any structure in the data at all. Then every query would have to touch all the data. The other extreme would be to sort all data which effectively provides an implicit structure. But then updates get expensive. Overall, there is no perfect design.

Tradeoffs in each structure

[S. Idreos, 2019]
New Applications Demand Change

**NEW APPLICATIONS**

existing systems need to change too

---

**ADAPT**
- WORKLOAD
- HARDWARE

**REASON**
- IMPROVE WITHIN A BUDGET
- WHAT WILL BREAK MY SYSTEM?

---

We increasingly need to think of new data structure designs, because applications and data change rapidly and because for data driven applications great performance comes only after rethinking the storage layer as well.
Many efforts in the field have been motivated by the vision of generating tailored systems for a specific scenario. In fact, even traditional databases are architected with this vision in mind. A generic database system can optimize a plan on the fly to match the query needs, it can choose from different storage and indexing options, etc. This is how generic database systems can be used in a wealth of applications! And then recent research has tried to push the boundaries of tailored designs by rethinking parts of the stack of a database system.
As a first step in this direction, we built an engine, which we call the Data Calculator and which takes as input the hardware, workload and layout of a data structure. It then computes automatically the algorithms that this data structure design needs to optimally process the workload on this hardware and it also computes the performance. That is, the response time that an actual implementation of this design would need to run this workload on this hardware. However, all this happens without the user having to implement anything and without even needing access to the actual hardware. Given this engine we show that we can start thinking about game-changing paradigms for system designs such as interactive design, self-designing systems, and fully automatic design for instance optimal systems.

Self-designing systems

[S. Idreos, 2019]
SageDB: a learned database system

Learned Data Structures and Algorithms
Discussion

• Is this the future?
• What about comparison baselines?
• Lots of work being done in this area
Benchmarking Learned Indexes

Figure 7: Performance and size tradeoffs for four different datasets. The black horizontal line represents the performance of binary search (which has a size of zero). Extended plots with all techniques are available here: https://rm.cab/lis1

Table 2: The fastest variant of each index structure compared against two hashing techniques on the amzn dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ns)</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>326.48</td>
<td>14.0</td>
</tr>
<tr>
<td>RS</td>
<td>266.58</td>
<td>4.0</td>
</tr>
<tr>
<td>RMI</td>
<td>180.90</td>
<td>48.0</td>
</tr>
<tr>
<td>BTree</td>
<td>482.11</td>
<td>166.0</td>
</tr>
<tr>
<td>IBTree</td>
<td>446.55</td>
<td>9.0</td>
</tr>
<tr>
<td>FAST</td>
<td>435.33</td>
<td>102.0</td>
</tr>
<tr>
<td>BS</td>
<td>741.69</td>
<td>0.0</td>
</tr>
<tr>
<td>CuckooMap</td>
<td>114.50</td>
<td>1541.0</td>
</tr>
<tr>
<td>RobinHash</td>
<td>93.69</td>
<td>6144.0</td>
</tr>
</tbody>
</table>

Table 2 lists the size and lookup performance of the best-performing (and thus often largest) variant of each index structure and both hashing techniques for a 32-bit version of the amzn dataset (results similar for others). Unsurprisingly, both hashing techniques offer superior point-lookup latency compared to traditional and learned index structures. This decreased latency comes at the cost of a larger in-memory footprint. For example, CuckooMap provides a 114ns lookup time compared to the 180ns provided by the RMI, but CuckooMap uses over 1GB of memory, whereas the RMI uses only 48MB. When range lookups and memory footprint are not concerns, hashing is a clear choice.

4.2.1 Larger datasets.
Figure 9 shows the performance / size tradeoff for each learned structure and a BTree for four different data sizes of the amzn dataset, ranging from 200M to 800M. All three learned structures are capable of scaling to larger dataset sizes, with only a logarithmic slowdown (as expected from the final binary search step). For example, consider an RMI that produces an average search bound that spans 128 keys, requiring 7 steps of binary search. If the dataset size doubles, an RMI of equal size is likely to return bounds that are twice as large: search bounds that span 256 keys. Such a bound requires only 8 total (1 additional) binary search steps. Thus, learned index structures scale to larger datasets in much the same way as BTrees. If larger datasets have more pronounced patterns, learned index structures may provide better scaling.

4.2.2 32-bit datasets.
Here, we scale down the amzn dataset from 64 to 32 bits, and compare the performance of the three learned index structures, BTrees, and FAST. The results are plotted in Figure 10. For learned structures, the performance on 32-bit data is nearly identical to performance on 64-bit data. Our implementations of RS and RMI both transform query keys to 64-bit floats, so this is not surprising. We attempted to perform computations on 32-bit keys using 32-bit floats, but found that the decreased precision caused floating point errors. The PGM implementation uses 32-bit computations for 32-bit inputs, achieving modest performance gains. For both tree structures, the switch from 64-bit to 32-bit keys allows twice as many keys to fit into a single cache line, improving performance. For FAST, which makes heavy use of AVX-512 operations, doubling the number of keys per cache line essentially doubles computational throughput as well, as each operator can work on 16 32-bit values simultaneously (as opposed to 8 64-bit values).

4.2.3 Search function.
Normally, we use binary search to locate the correct key within the search bound provided by the index. However, other search techniques can be used. Figure 11 evaluates using binary, linear, and interpolation search for various index structures on osm and amzn. We observed that binary search (first column) was always faster than linear search (second column). This aligns with prior work that showed binary search being effective until the data size dropped below a very small threshold. 

[R. Marcus et al., 2021]
We evaluate Tsunami against Flood, the original in-memory learned multi-dimensional index, as well as a number of traditional non-learned indexes, on a variety of workloads over real datasets. We show that Tsunami is up to 6× and 11× faster than Flood and the fastest optimally-tuned non-learned index, respectively. Tsunami is also adaptable to workload shift, and scales across data size, query selectivity, and dimensionality.

In the remainder of this paper, we give background (§2), present an overview of Tsunami (§3), introduce its two core components—Grid Tree (§4) and Augmented Grid (§5), present experimental results (§6), review related work (§7), propose future work (§8), and conclude (§9).

2 BACKGROUND

Tsunami is an in-memory clustered multi-dimensional index for a single table. Tsunami aims to increase the throughput performance of analytics queries by decreasing the time needed to filter records based on range predicates. Tsunami supports queries such as:

\[
\text{SELECT SUM(R.X) FROM MyTable WHERE (a \leq R.Y \leq b) AND (c \leq R.Z \leq d)}
\]

where \(\text{SUM(R.X)}\) can be replaced by any aggregation. Records in a \(d\)-dimensional table can be represented as points in \(d\)-dimensional data space. For the rest of this paper, we use the terms record and point interchangeably. To place Tsunami in context, we first describe the k-d tree as an example of a traditional non-learned multi-dimensional index, and Flood, which originally proposed the idea of learned in-memory multi-dimensional indexing.

2.1 K-d Tree: A Traditional Non-Learned Index

The k-d tree \([4]\) is a binary space-partitioning tree that recursively splits \(d\)-dimensional space based on the median value along each dimension, until the number of points in each leaf region falls below a threshold, called the page size. Fig. 1a shows a k-d tree over 2-dimensional data that has 8 leaf regions. The points within each region are stored contiguously in physical storage (e.g., a column store). By construction, the leaf regions have a roughly equal number of points. To process a query (i.e., identify all points that match the query’s filter predicates), the k-d tree traverses the tree to find all leaf regions that intersect the query’s filter, then scans all points within those regions to identify points that match the filter predicates.

The k-d tree structure is constructed based on the data distribution but independently of the query workload. That is, regardless of whether a region of the space is never queried or whether queries are more selective in some dimensions than others, the k-d tree would still build an index over all data points with the same page size and index overhead. While other traditional multi-dimensional indexes split space in different ways \([3, 27, 31, 47]\), they all share the property that the index is constructed independent of the query workload.

2.2 Flood: A Learned Index

In contrast, Flood \([30]\) does optimize its layout based on the workload (see Fig. 1b). We first introduce how Flood works, then explain its two key advantages over traditional indexes, then discuss two key limitations it has.

[Diagram showing K-d Tree, Flood, and Tsunami]
Query Optimization

Parser

SQL

Query Optimizer

Hint set 1

Hint set 2

... 

Hint set 3

TCNN

Reward
Predictions

Execution Engine

Training

Experience

User provided

Query plan

External component

Bao

[R. Marcus et al., 2021]
Reminders

• Final Exam Review Wednesday (come with questions!)
• Final Exam on Monday, May 9 from 4-5:50pm