Report from Dagstuhl Seminar 22111

Database Indexing and Query Processing

Renata Borovica-Gajic*[1], Goetz Graefe*[2], Allison Lee*[3], Caetano Sauer*[4], and Pinar Tözün*[5]

1 **The University of Melbourne, AU.** renata.borovica@unimelb.edu.au
2 **Google – Madison, US.** goetz.graefe@gmail.com
3 **Snowflake – San Mateo, US.** allison@snowflake.com
4 **Salesforce – München, DE.** caetano.sauer@salesforce.com
5 **IT University of Copenhagen, DK.** pito@itu.dk

Abstract

The Dagstuhl Seminar 22111 on "Database indexing and query processing", held from March 13 to March 18 2022, brought together researchers from academia and industry to discuss robustness in database management systems. This seminar was a continuation of previous seminars on the topic of Robust Query Processing, where we included *indexing* as a general topic and also discussed aspects that have not been addressed by the previous instances of the seminar. This article summarizes the main discussion topics, and presents the summary of the outputs of three work groups that discussed: i) storage architectures, ii) robust operators, and iii) indexing for data warehousing.

Seminar March 13–18, 2022 – http://www.dagstuhl.de/22111
2012 ACM Subject Classification Information systems → Data management systems; Information systems → Storage architectures
Keywords and phrases database, execution, hardware, optimization, performance, query
Digital Object Identifier 10.4230/DagRep.12.3.82

1 Executive Summary

Renata Borovica-Gajic (The University of Melbourne, AU)
Goetz Graefe (Google – Madison, US)
Allison Lee (Snowflake – San Mateo, US)
Caetano Sauer (Salesforce – München, DE)
Pinar Tözün (IT University of Copenhagen, DK)

License Creative Commons BY 4.0 International license
© Renata Borovica-Gajic, Goetz Graefe, Allison Lee, Caetano Sauer, and Pinar Tözün

The Dagstuhl Seminar 22111 on "Database indexing and query processing" assembled researchers from industry and academia for the fourth time to discuss robustness issues in database query performance. The seminar gathered researchers around the world working on indexing, storage, plan generation and plan execution in database query processing, and in cloud-based massively parallel systems with the purpose to address the open research challenges with respect to the robustness of database management systems. Delivering robust query performance is well known to be a difficult problem for database management systems. All experienced DBAs and database users are familiar with sudden disruptions in data centers due to poor performance of queries that have performed perfectly well in the past. The goal of the seminar was to discuss the current state-of-the-art, to identify specific research opportunities in order to improve the state-of-affairs in query processing, and to develop new

* Editor / Organizer

Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license
Database Indexing and Query Processing, *Dagstuhl Reports*, Vol. 12, Issue 3, pp. 82–96
Editors: Renata Borovica-Gajic, Goetz Graefe, Allison Lee, Caetano Sauer, and Pinar Tözün
Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

approaches or even solutions for these opportunities, building upon successes of the past Dagstuhl Seminars [1, 2, 3]. The organizers (Renata Borovica-Gajic, Goetz Graefe, Allison Lee, Caetano Sauer, and Pinar Tözün) this time attempted to have a focused subset of topics that the participants discussed and analyzed in more depth. From the proposed topics on algorithm choices, join sequences, learned and lightweight indexes, database utilities, modern storage hardware, and benchmarking for robust query processing, the participants formed three work groups: i) one discussing indexing for data warehousing, ii) one discussing robust query operators, and iii) one discussing robust storage architectures. Upon choosing the topics of interest, the organizers then guided the participants to approach the topic through a set of steps: by first considering related work in the area; then introducing metrics and tests that will be used for testing the validity and robustness of the solution; after metrics, the focus was on proposing specific mechanisms for the proposed approaches; and finally the last step focused on the implementation policies. At the end of the week, each group presented their progress with the hope to continue their work towards a research publication. The reports of work groups are presented next.

References

1 Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth scan: Statistics-oblivious access paths. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *ICDE*, pages 315–326. IEEE Computer Society, 2015.

2 Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth scan: robust access path selection without cardinality estimation. *VLDB J.*, 27(4):521–545, 2018.

3 Martin L. Kersten, Alfons Kemper, Volker Markl, Anisoara Nica, Meikel Poess, and Kai-Uwe Sattler. Tractor pulling on data warehouses. In Goetz Graefe and Kenneth Salem, editors, *DBTest*, page 7. ACM, 2011.

2 Table of Contents

Executive Summary
Renata Borovica-Gajic, Goetz Graefe, Allison Lee, Caetano Sauer, and Pinar Tözün 82

Working Groups

Storage Architectures
Pinar Tözün, Goetz Graefe, Thomas Heinis, Sangjin Lee, Alberto Lerner, Danica Porobic, Daniel Ritter, Lukas Vogel, and Tianzheng Wang 85

Robust Operators
Allison Lee, Carsten Binnig, Thomas Bodner, Campbell Fraser, Mhd Yamen Haddad, David Justen, Andrew Lamb, Bart Samwel, and Nga Tran 89

Indexing for Data Warehousing
Caetano Sauer, Peter A. Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Knut Stolze, and Marcin Zukowski 92

Participants 96

Remote Participants 96

3 Working Groups

3.1 Storage Architectures

Pinar Tözün (IT University of Copenhagen, DK), Goetz Graefe (Google – Madison, US), Thomas Heinis (Imperial College London, GB), Sangjin Lee (Hanyang University – Seoul, KR), Alberto Lerner (University of Fribourg, CH), Danica Porobic (Oracle Switzerland – Zürich, CH), Daniel Ritter (Hasso-Plattner-Institut, Universität Potsdam, DE), Lukas Vogel (TU München, DE), and Tianzheng Wang (Simon Fraser University – Burnaby, CA)

License Ⓒ Creative Commons BY 4.0 International license
© Pinar Tözün, Goetz Graefe, Thomas Heinis, Sangjin Lee, Alberto Lerner, Danica Porobic, Daniel Ritter, Lukas Vogel, and Tianzheng Wang

The storage hierarchy has been getting deeper and more heterogeneous. In addition, platforms that enable computational storage and/or near-data processing are becoming more widely-available [1]. This storage landscape is an opportunity for data-intensive systems. However, it also presents us with several challenges when it comes to exploiting these technologies.

One key challenge that comes with the deeper and heterogeneous storage landscape is the various sources of unpredictability.

- Device types: Hard disks (HDD), Solid-state Drives (SSD), Persistent Memory (PMEM), DRAM have different device characteristics requiring the end-users to adopt different system optimizations. In addition, there may even be variety among the same class of devices. For example, SSDs are not a uniform class of devices. There are space-optimized (QLC, TLC) or speed-optimized (SLC) SSDs, and devices from different vendors behave differently.
- Interfaces: With the variety of the devices comes also the variety of device interfaces to interact with. Even within the same class of devices, there could be different options. For example, NVMe standard defines different interfaces for key-value SSDs, zoned-namespaces, computational storage (currently being standardized), etc.
- Disaggregated storage: It is common practice to separate compute nodes from storage nodes for large-scale hardware deployments. Accessing a locally-attached storage device could behave differently than accessing a storage device over the network.
- Access modes: There are different ways to access storage devices. One may include CPU on the path or bypass it using direct memory access (DMA). Some accesses may be transparent to the end-user implicitly being controlled by hardware itself, while some hardware may give more explicit controls to the end-user for application-specific optimizations.
- Workloads: Data-intensive workloads exhibit a high variety as well. While some workloads have well-behaved and predictable data read/write and movement characteristics, some can have unpredictable ad-hoc behavior.
- Infrastructure: Today many data-intensive systems run on the cloud. Cloud infrastructures take away the burden of managing a big hardware infrastructure from the end-users. However, they do so by abstracting or virtualizing hardware. This means that servers and storage devices used by a data-intensive system may change at any point. In addition, servers from different popular vendors that make up the cloud support different technologies. For example, Intel servers come with support for persistent memory, while AMD servers don't have this support. In contrast, AMD servers are optimized for supporting many PCIe lanes making them good choices if one wants to deploy many SSDs.

In the storage working group of this Dagstuhl Seminar, we specifically focused on the following research question: How can we robustly exploit the modern storage hierarchy despite all the sources of unpredictability?

If one digs deeper, at the heart of this challenge lies the challenge of orchestrating the data movement across the variety storage layers and devices. Therefore, the question above boils down to how can we orchestrate the data movement across layers to get more predictably good performance (a) when a workload is well-behaved and (b) when a workload isn't well-behaved?

3.1.1 Well-behaved scenario

We started our discussion focusing on the easier case, which is the well-behaved scenario. A representative workload for this scenario is external sort, which is a building block for many data-intensive operations such as the compaction operation for log-structured merge trees, sorting results of a query, etc. The key challenge with this operation is the temporary data it creates, which in turn creates storage pressure. Our goal is to design a robust and efficient external sort mechanism that can exploit different storage layers. The main design principle / intuition of our mechanism is to **separate the read and write traffic for the data movement**.

While we aimed at avoiding any assumptions regarding the functionality of available storage devices, one key requirement for the efficiency of our mechanism is having a form of DMA support. This is not an unreasonable requirement for today's storage landscape considering the availability of PCIe DMA engine for SSDs, ioat for moving data from DRAM to PMEM, S3 async put in the cloud, remote direct memory access (RDMA), etc.

Next we describe the external sort mechanism following our goal and design principle. There are two versions of it that differ in the way the sort and merge tasks are scheduled. Each version also has an associated illustration.

Way up / Sorts: The data to be sorted is read directly to processor caches from the persistent storage unit, which is the data source, using the DMA engine. The size unit of these fetches, let's call them runs, could be based on the LLC cache size per core divided by 2. The reason is for each direct data fetch to cache, even though one bypasses the CPU and memory layers, the associated memory space has to be allocated. We need twice the space to allow dual-buffering at LLC rather than going to DRAM while a core is sorting the fetched data.

Way down / Merges: In the non-pipelined version, we first wait for all data to be sorted in units of runs before each core starts merging of these runs.

Each core performs merges till the DRAM size is exhausted. The merge-sorted run can be spilled to a persistent storage device as soon as the initial block/page of it is produced. This persistent storage device could be a different one than the data source if such a device is available. We will call it the staging area. Ideally, such a staging area should have low access latency such as PMEM or new-gen SSDs. Ideal number of runs a core merges at a time still requires a discussion.

The merge-sorted runs are read from the staging area using DMA using a fetch unit similar to the way up / sorting phase. However, this time, the runs are already sorted, so the cores just perform merging. This is repeated as long as it is needed to get the final merge-sorted run.

Where or which device we end up writing the sorted run to depends on the use case.

The main difference between the non-pipelined and the pipelined mechanisms is the way in which available cores are assigned to sort and merge tasks of the external merge-sort task.

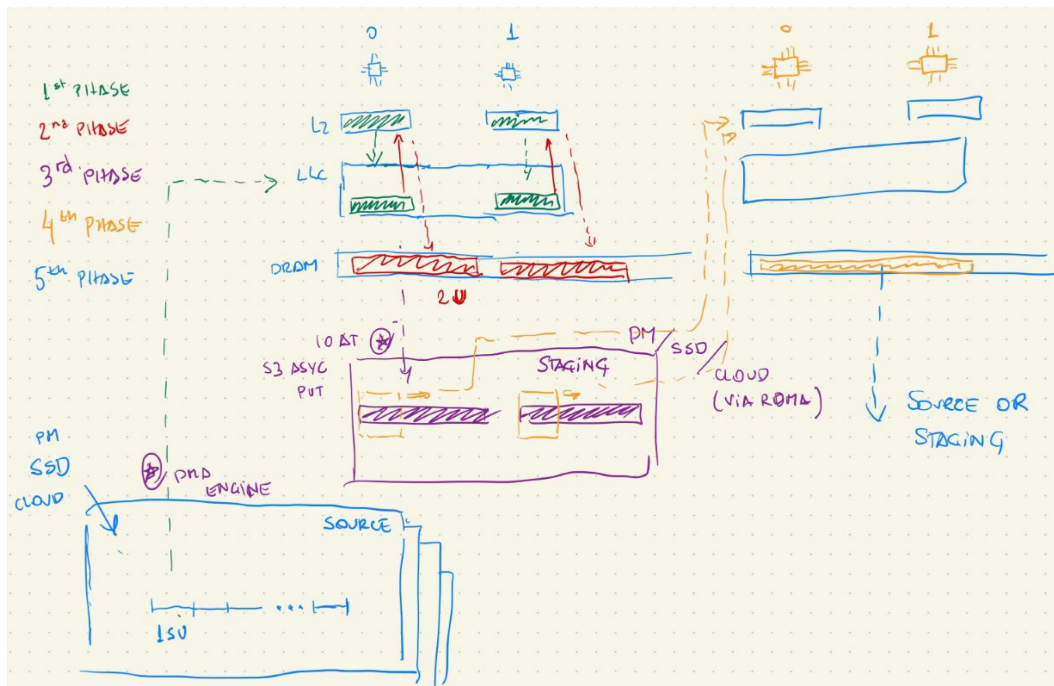

Figure 1 Not-pipelined scenario.

In the non-pipelined merge-sort mechanism, the sort (way-up) and merge (way-down) stages are separate stages. First, all the available cores sort the runs and then they all merge the sorted runs. Rather than this strict separation of the two stages, one can assign some cores for sorting the runs and some cores for merging, where the sorted runs are transferred to the cores responsible for merging. In this scheme the sort and merge operations are pipelined in stages, similar to earlier work like StagedDB and SharedDB.

How the data is fetched from or written to persistent storage devices is the same across the not-pipelined and pipelined versions of the merge-sort.

3.1.2 Not well-behaved scenario

During the seminar, we didn't have time to talk in detail about the not well-behaved scenarios. Such scenarios are characterized by the unpredictability of the read and write patterns such as online transaction processing (OLTP) workloads.

In the literature, a common way to handle OLTP workloads is creating hardware-conscious data structures such as log-structured merge trees [3], B-epsilon tree [2], Plush [4], Apex [5], etc. The main design goal when it comes to creating these data structures is to morph the workload's unpredictable data access patterns or movement to a more well-behaved pattern for the target storage device. The issue is that usually there is only one or two devices targeted such as DRAM & PMEM and DRAM & SSD. There are only a few recent works (e.g., Umzi [6], NovaLSM [7], etc.) that target multiple layers of storage hierarchy or disaggregated storage.

We overall support the approach of morphing the data movement patterns using novel and hardware-conscious data structures for not well-behaved workloads. On the other hand, we encourage our research community to consider the new and multiple layers of the storage hierarchy when adopting this approach.

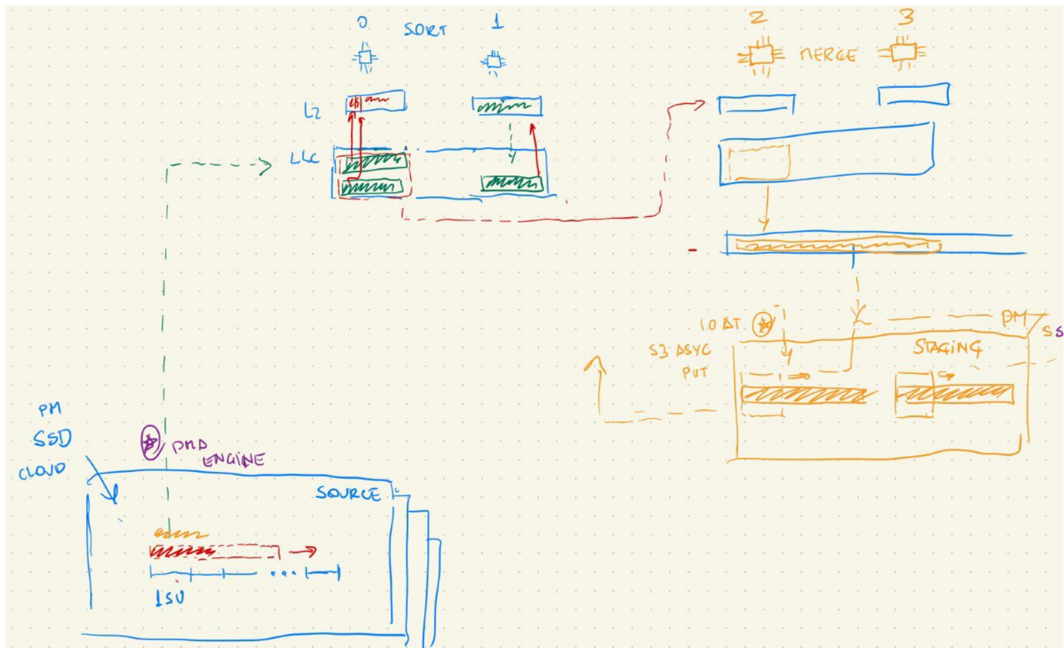

Figure 2 Pipelined scenario.

3.1.3 Co-design of storage and data-intensive systems

During the seminar, we also didn't have the time to touch upon challenges for co-design and utilizing computational storage for data-intensive systems.

The co-design challenge boils down to the trade-off between having a predictable but sub-optimal mechanism vs unpredictable but smart mechanism. It is easier to have co-design principles for well-behaved workloads that would lead to predictably smart and optimal choices. However, the not well-behaved patterns may lead to unpredictability, which may overweigh the gains of being smart and optimal most of the time when interacting with storage devices. Nevertheless, it is still worthwhile to deploy storage and data-intensive system co-design mechanisms for operations such as filtering, encryption, compression, etc.

3.1.4 Next Steps

The next steps to this work are:
- Modeling the data movement cost to reason about benefits
- Reasoning about the tuning of parameters such as data fetch units, degree of parallelism, number of runs to merge, etc.
- Discussion on what happens if the server is shared with other requests
- Implementing the two versions of the external sort mechanism
- Additional work orthogonal to external sort design: extensive storage access tracing for big database systems

References

1 Alberto Lerner and Philippe Bonnet. Not your grandpa's SSD: the era of co-designed storage devices. In *SIGMOD*, pages 2852–2858, 2021.

2 Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to bϵ-trees and write-optimization. *login Usenix Mag.*, 40(5), 2015.
3 Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
4 Under submission/review.
5 Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. APEX: A high-performance learned index on persistent memory. *Proc. VLDB Endow.*, 15(3):597–610, 2021.
6 Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. Umzi: Unified multi-zone indexing for large-scale HTAP. In *EDBT*, pages 1–12, 2019.
7 Haoyu Huang and Shahram Ghandeharizadeh. Nova-lsm: A distributed, component-based lsm-tree key-value store. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 749–763. ACM, 2021.

3.2 Robust Operators

Allison Lee (Snowflake – San Mateo, US), Carsten Binnig (TU Darmstadt, DE), Thomas Bodner (Hasso-Plattner-Institut, Universität Potsdam, DE), Campbell Fraser (Google – Mountain View, US), Mhd Yamen Haddad (Ecole Polytechnique – Palaiseau, FR), David Justen (Hasso-Plattner-Institut, Universität Potsdam, DE), Andrew Lamb (InfluxData – Boston, US), Bart Samwel (Databricks – Amsterdam, NL), and Nga Tran (InfluxData – Boston, US)

License Creative Commons BY 4.0 International license
© Allison Lee, Carsten Binnig, Thomas Bodner, Campbell Fraser, Mhd Yamen Haddad, David Justen, Andrew Lamb, Bart Samwel, and Nga Tran

The Dagstuhl Seminar 17222 on "Robust Performance in Database Query Processing" proposed a novel dynamic join order selection path method named "Plan of Least Resistance", which is described in the Dynamic Join Sequence working group section of the seminar report [1].

This novel algorithm aimed to increase the robustness of query processing by dynamically avoiding poorly chosen join orders based on runtime feedback. However it was not clear after the conclusion of Seminar 17222 how widely applicable and implementable this novel algorithm is.

Research Question: Is the "Plan of Least Resistance" (POLR) approach for robust query processing practical for commercial systems?

Success definition: Outline a minimal commercially viable implementation of POLR.

3.2.1 Review: Plan of Least Resistance

There are many open questions to this approach, so we focused only on those that must be resolved for a minimum viable commercial implementation of this approach:

- What shapes/orders of join plans are possible in the potentially routable paths, and which possible join orders should be included in the plan?
- What is the routing policy for the Multiplexer, and what cost metrics are required to implement that policy?

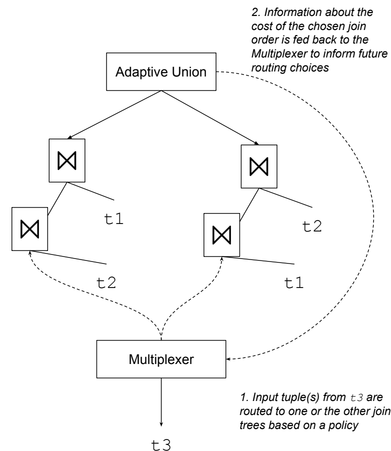

Figure 3 Schematic Plan of Least Resistance: a Router chooses based on some model which of two join orders to route tuples from t3. One path joins t2 first and then t3, and one path joins t3 first followed by t2. The execution engine tracks the cost of evaluating the join tree that was chosen, for the tuple(s) and feeds that cost information back to the router to inform its future choices.

3.2.2 Join Order Selection

The goal of Join Order selection is to determine a practical way to pick join orders that will provide robust query performance for the switcher.

Assumptions:

- System that will use hash join with some form of sideways information passing (SIPS) filter that can be applied to join keys during scans of other relations
- Only consider linear join plans (left deep / right deep depending on which side you prefer to draw the build input)

Using the information about the input relation sizes, the system leverages a classical optimizer to pick a candidate set of plans. An algorithm was proposed to generate a set of plans that provides good coverage across the space of possible cardinalities of intermediate results. The initial table to scan can also be chosen with the help of the classical optimizer, to pick the best cost table that is robust across the space of cardinalities.

This approach is more robust than a fixed join order, and it is implementable in typical commercial database systems.

3.2.3 Multiplexer Tuple Routing

The goal of the router is to route input tuples the best among available join orders. Its dynamic nature also allows it to adapt automatically to changes in the input and be robust to various clusterings of input values in the incoming data stream.

We propose a “bounded regret” approach algorithm to select which possibility a particular tuple is routed to. Specifically, the user provides a budget for how much extra time the robust plan to spend vs. the fastest currently known plan. The router will then choose a tuple routing to stay within this budget.

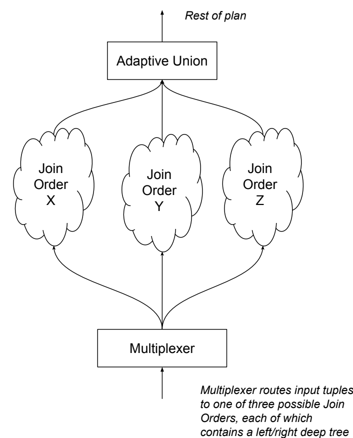

Figure 4 A Multiplexer beneath an Adaptive Union routes tuples to one of three possible join orders.

Initially the router will send "enough" tuples to all three of the branches to be confident in the observed cost. After this initial phase, the router will send tuples to each join order with a certain probability, depending on the observed cost of that plan, in order to constrain the overhead to within the budget, while maximizing observations of potentially better plans. If the observed costs change significantly over time, then this algorithm is run again to update the weights.

This technique is more robust than picking the best order after initial measurement because it can switch between multiple plans over time, and even if it gets it wrong initially the runtime feedback loop can guide it to a better plan over time. This technique is implementable as it requires straightforward calculations that are neither overly burdensome to implement and require trivial CPU and memory resources, and are easy to test.

3.2.4 Next Steps

During our week at Dagstuhl, we proposed a practical, robust solution to join order selection in database systems. The next steps for this work include:

- Build a research prototype of our solution. This would allow us to experiment with some of the alternative policy options that we considered for join order generation and tuple routing.
- Propose solutions to open questions unrelated to the minimal implementation, including different join shapes, distributed execution plans, and spilling operators.

References

1 Renata Borovica-Gajic, Goetz Graefe, and Allison W. Lee. Robust performance in database query processing (dagstuhl seminar 17222). *Dagstuhl Reports*, 7(5):169–180, 2017.

3.3 Indexing for Data Warehousing

Caetano Sauer (Salesforce – München, DE), Peter A. Boncz (CWI – Amsterdam, NL), Yannis Chronis (University of Wisconsin-Madison, US), Jan Finis (Salesforce – München, DE), Stefan Halfpap (Hasso-Plattner-Institut, Universität Potsdam, DE), Viktor Leis (Universität Erlangen-Nürnberg, DE), Thomas Neumann (TU München, DE), Anisoara Nica (SAP SE – Waterloo, CA), Knut Stolze (IBM Deutschland – Böblingen, DE), and Marcin Zukowski (Snowflake – San Mateo, US)

License Creative Commons BY 4.0 International license
© Caetano Sauer, Peter A. Boncz, Yannis Chronis, Jan Finis, Stefan Halfpap, Viktor Leis, Thomas Neumann, Anisoara Nica, Knut Stolze, and Marcin Zukowski

Selective queries are quite common in large-scale data analytics; for example, when drilling down into a specific customer in a dashboard. Traditionally, selective queries are optimized by creating secondary indexes. However, because of their large size, expensive maintenance, and difficulty to tune and automate, indexes are typically not used in modern cloud data warehouses. Instead, such systems rely mostly on full table scans and lightweight optimizations like min/max filtering, whose effectiveness depends heavily on the data layout and value distributions. It is also difficult to predict whether certain columns will be targeted by selective queries or not, which may preclude an upfront decision to create indexes.

In this working group, we sketched a general indexing framework called SPA (Smooth Predicate Acceleration). It optimizes selective queries automatically, by adaptively indexing subsets of data in an incremental and workload-driven manner. It makes fine-granular decisions and continuously monitors their benefit, dynamically allocating an optimization budget in a way that bounds the additional cost of indexing. Furthermore, it guarantees a performance improvement in the cases where indexes—potentially partial ones—prove to be beneficial. On the other hand, when indexes lose their benefit due to a shifting workload, they are also gradually deconstructed in favor of optimizations that accommodate recent trends.

3.3.1 Desiderata

The framework envisioned in our working group should be:

- Workload-driven: indexes are created and dropped solely based on workload observations, without upfront decisions or manual interventions.
- Smooth: index maintenance is carried out in incremental steps, as a side-effect of table scans and without spikes in query latency.
- Economical: decisions are taken and evaluated based solely on the monetary cost in comparison to a baseline of full table scans.
- Cost-bounded (i.e., “do no harm”): bad decisions should not impact the user-observed response times by more than a configurable percentage.
- Modular: the framework supports different types of index or summary structures, and their individual characteristics are taken into account by the economic model.

3.3.2 General approach

The SPA framework observes the workload and automatically maintains partial indexes in an incremental manner. The decisions taken by the framework are purely economical: it tracks the additional cost of index maintenance (for both computing and storage) as well as the benefit provided by indexes during scans. A positive balance on this benefit gives

the framework more budget to continue building indexes; a negative balance, on the other hand, leads to a gradual deconstruction of indexes. Thus, index creation can be seen as an investment with continuously evaluated returns. The additional cost of indexing is bounded thanks to a configurable budget (or “indexing tax”), which is specified as a percentage of the cost of a full table scan (e.g., 1%): if none of the indexing investments pay off, the system guarantees that queries will not be slowed down by more than this percentage on average.

Indexes are built incrementally by first indexing individual subsets of a table, such as a file or a block on storage. These are considered units of scanning which can be skipped with available summaries such as min/max small materialized aggregates (SMAs) [2, 1]. If the available summaries are not able to filter out a particular block and matches are not found for a given predicate, then the SPA framework might create an index on that block specifically. On subsequent scans, that index can be probed before fetching and scanning the corresponding block. As more and more blocks get indexed, they might also be merged into larger indexes covering multiple blocks, similar to a log-structured merge tree. These partial index structures might also lose their value over time, in which case a caching policy can drop them or deconstruct large indexes into smaller ones.

3.3.3 Simulation

To simulate the behavior of SPA, we implemented a mock of an in-memory, column-oriented table scan operator in C++. This prototype, available in an open source repository[1], organizes records into blocks of 100,00 tuples. It uses a simple randomized approach to create indexes in these blocks individually. This works as follows: whenever a block is scanned and no match is found for the simulated predicate, the system randomly chooses whether to create an index for this block. The probability of this choice is proportional to an artificial budget variable. This variable is incremented by a small fixed amount with every block scanned (1), and decremented by a much larger amount if an index is created (2). In the case where an index is available and this index allows a block to be skipped, the budget is increased by a comparatively large amount (3). The reasoning behind each of these budget changes is explained below, referring back to the numbers in parentheses above:

1. A small budget should be accrued regularly to allow for index creations in the first place; this can be seen as a regular small deposit (or savings) into the index maintenance account.
2. Creating an index has a non-negligible cost on scan performance; it is an investment that decreases the account balance but hopefully brings returns in the future.
3. If an index allows the scan operator to skip blocks, then the investment has paid off, and returns are deposited into the account.

As more budget is accrued (hopefully exponentially thanks to compounded returns) and more indexes are built, smaller indexes are also merged into larger ones. Just like index creation, the merge operation also deducts from the budget and pays back returns whenever it is used to avoid scan work.

Figure 5 below shows the observed query response times from an execution of this prototype with different deposit rates, i.e., the budget increase with every block scan in step 1 above. Note that this is a unitless parameter, as it just serves to scale the probability of creating an index. This experiment sends repeated queries (x axis) with a random equality predicate on a given column. The query response time is plotted in the y axis.

[1] https://github.com/JFinis/dagstuhl-spa

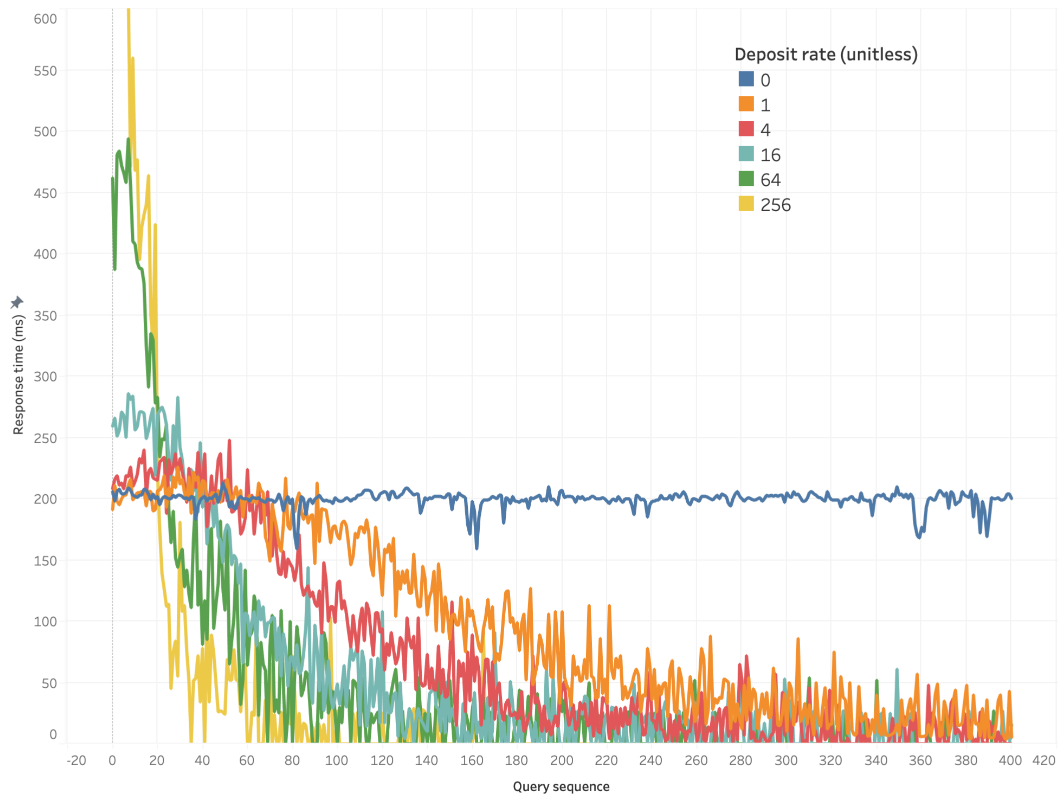

Figure 5 Simulation of budget-driven index creation.

As the results show, a deposit rate of zero (blue line) has nearly constant response time of 200ms, serving as the baseline for the experiment. As the deposit rate increases, the first queries in the sequence observe larger response times, but they converge faster into a fully index-based scan, with response time under 50ms. This reflects the expected behavior of our economic model: lower deposit rates have lower disturbance in query response times, while higher deposit rates benefit faster from indexing performance; in the end, all choices converge to faster execution speeds thanks to indexing.

3.3.4 Future work

Our working group considers the ideas developed during this seminar novel and industry-relevant. As such, we plan to refine these ideas into a more detailed description of the SPA framework and submit them as part of a vision paper to a major database conference. To validate the benefits investigated with the prototype implementation described above, we also plan to implement a cost-based prototype in a commercial database system and publish our evaluation results as part of the aforementioned vision paper.

On the technical side, there are also multiple avenues to pursue in terms of design choices:

- Experiment with different index structures, which might trade-off accuracy for space consumption.
- Evaluate partial index structures in terms of how efficient and simple they are to merge and deconstruct incrementally (i.e., their composability).
- Investigate different cost models, especially focused on the cost of resources in the cloud.

References

1. Goetz Graefe. Fast loads and fast queries. In Torben Bach Pedersen, Mukesh K. Mohania, and A Min Tjoa, editors, *DaWaK*, volume 5691 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 2009.
2. Guido Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487. Morgan Kaufmann, 1998.

Participants

- Carsten Binnig
TU Darmstadt, DE
- Thomas Bodner
Hasso-Plattner-Institut, Universität Potsdam, DE
- Peter A. Boncz
CWI – Amsterdam, NL
- Yannis Chronis
University of Wisconsin-Madison, US
- Jan Finis
Salesforce – München, DE
- Campbell Fraser
Google – Mountain View, US
- Mhd Yamen Haddad
Ecole Polytechnique – Palaiseau, FR
- Stefan Halfpap
Hasso-Plattner-Institut, Universität Potsdam, DE
- Thomas Heinis
Imperial College London, GB
- David Justen
Hasso-Plattner-Institut, Universität Potsdam, DE
- Andrew Lamb
InfluxData – Boston, US
- Allison Lee
Snowflake – San Mateo, US
- Sangjin Lee
Hanyang University – Seoul, KR
- Viktor Leis
Universität Erlangen-Nürnberg, DE
- Alberto Lerner
University of Fribourg, CH
- Thomas Neumann
TU München, DE
- Anisoara Nica
SAP SE – Waterloo, CA
- Danica Porobic
Oracle Switzerland – Zürich, CH
- Daniel Ritter
Hasso-Plattner-Institut, Universität Potsdam, DE
- Bart Samwel
Databricks – Amsterdam, NL
- Caetano Sauer
Salesforce – München, DE
- Knut Stolze
IBM Deutschland – Böblingen, DE
- Pinar Tözün
IT University of Copenhagen, DK
- Nga Tran
InfluxData – Boston, US
- Lukas Vogel
TU München, DE
- Tianzheng Wang
Simon Fraser University – Burnaby, CA
- Marcin Zukowski
Snowflake – San Mateo, US

Remote Participants

- Renata Borovica-Gajic
The University of Melbourne, AU
- Goetz Graefe
Google – Madison, US