# SPA: Economical and Workload-Driven Indexing for Data Analytics in the Cloud

Peter Boncz boncz@cwi.nl
Yannis Chronis chronis@cs.wisc.edu
Jan Finis jfinis@salesforce.com
Stefan Halfpap stefan.halfpap@hpi.de
Viktor Leis leis@in.tum.de
Thomas Neumann neumann@in.tum.de
Anisoara Nica anisoara.nica@sap.com
Caetano Sauer caetano.sauer@salesforce.com
Knut Stolze stolze@de.ibm.com
Marcin Zukowski marcin.zukowski@snowflake.com

***Abstract*—Selective queries are not uncommon in large-scale data analytics, for example, when drilling down into a specific customer in a dashboard. Traditionally, selective queries are accelerated 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 or data lakes. 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.**

**We propose SPA as the vision for automatically optimizing selective queries for immutable copy-on-write data formats. SPA adaptively indexes subsets of the data in an incremental and workload-driven manner. It makes fine-grained 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. When indexes lose their benefit due to a shifting workload, they are gradually deconstructed in favor of optimizations that accommodate recent trends. As SPA does not require information about updates performed on the data, it can also be employed as an accelerator for systems that do not control the data, e.g., in cloud data lakes.**

## I. Introduction

**Indexing in cloud data warehouses.** For analytics on data lakes and warehouses, the conventional wisdom is that full-blown secondary indexes (e.g., B-trees) are not very useful. Creating a secondary index is time-consuming, and the additional storage cost may simply be too expensive for large data sets. Keeping indexes up-to-date is also costly, as tables are typically periodically appended or updated while shedding data using life-cycle management. It is also difficult to anticipate whether specific columns will be targeted by selective queries or not, which may preclude an upfront decision to create indexes. In cloud data lakes, there tends to be less oversight over workloads, and there may be no human database administrator (DBA) to make such decisions. Moreover, the system performing updates might not be the same system performing analytical queries. For these reasons, today's cloud analytics systems often miss optimization opportunities for selective queries, and workloads are unnecessarily expensive in these systems.

**Smooth Predicate Acceleration.** In this work, we propose *Smooth Predicate Acceleration* (*SPA*), a framework that optimizes queries with selective predicates in an automatic and adaptive way, guided by an economic model that can be tailored to cloud environments. It achieves this by incrementally constructing and deconstructing partial indexes based on their observed benefit at runtime. As a vision, our aim for SPA is not to propose one specific way to perform such optimizations. We rather raise the awareness that adaptive indexing based on economic models could yield unique benefits. This vision is especially attractive in the cloud, where compute and storage are virtually limitless. Therefore, an optimization that proves to be economical is always a valid one.

**Desiderata.** The framework envisioned in this paper is:

- Workload-driven: indexes are automatically constructed and deconstructed solely based on workload observations, without upfront decisions, manual DBA interventions, or knowledge of updates on the underlying data.
- Smooth: index maintenance is carried out in small incremental steps, as a side-effect of table scans and without spikes in query latency or long-running background tasks.
- Cost-bounded (i.e., "do no harm"): bad decisions should not impact the user-observed response times and monetary cost by more than a small configurable percentage.
- Economical: decisions are taken and evaluated based on the monetary cost in comparison to a baseline of full table scans.
- Modular: the framework does not prescribe the types of index or summary structures used, supports various economic models, and is configurable in the way indexes are maintained.

**General approach.** Without an index, every query has to perform an expensive ($O(n)$) full table scan. The SPA framework observes the workload and maintains partial indexes automatically and incrementally. The decisions taken by the framework are purely *economical*: it tracks the additional cost of index maintenance (for both compute and storage) and the benefit provided by indexes during scans. A positive balance on this benefit gives the framework more budget to continue constructing indexes; a negative balance, on the other hand, leads to a gradual deconstruction of indexes. Thus, index construction can be seen as an investment with continuously evaluated returns. The additional cost of indexing is bounded by a configurable budget (or "investment deposit"), which can

be 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 and their execution costs will not increase by more than this percentage on average. The cost-based nature of our approach also allows incorporating different kinds of index structures. The ideas behind SPA are applicable to different systems in different ways, which is why we present it as a general framework in a vision paper rather than a specific implementation for one particular system.

## II. Framework

This section describes the general operation of the SPA framework and how it achieves the five desiderata above.

### A. Preliminaries

**Use case.** This paper focuses on the use case of data analytics in the cloud, where data is stored on highly-available and elastic object stores like Amazon S3, potentially in open file formats like Apache Parquet [1] that are not necessarily under exclusive control of a single system (i.e., data lakes). Nevertheless, the framework is general enough to also be applicable to other database architectures, e.g., on-premise data warehouses or modern copy-on-write database systems [2], [3]. Queries—which are often exploratory or ad-hoc—contain arbitrary predicates that are pushed down into scan operators. Scans make use of basic filtering techniques (such as min/max synopses [4] or zone maps [5]) to skip data ranges that cannot satisfy the given predicates. Selective predicates where such early filtering is not very effective (e.g., because there is no correlation between the selected column and the order of tuples on storage) require full table scans, whose monetary cost can often be calculated (e.g., $5 per TB scanned for BigQuery or Athena).

**Physical layout.** We assume that tables are split horizontally into immutable chunks we hereinafter call *blocks*. Thus, any update will create a new block instead of mutating an existing one and therefore *an index on a given set of blocks always stays valid*. Many modern data warehouse and data lake technologies, but also general analytical or HTAP systems employ such a form of immutable blocks, making SPA applicable to a wide range of systems. Examples include proprietary systems such as Snowflake [6] or Hyper [2], open columnar file formats like Apache Parquet [1] or Apache ORC [7], open data-lake table formats such as DeltaLake [8] or Apache Iceberg [9], as well as Hyrise's chunk architecture [3]. These formats may have different interpretations and granularities for what constitutes a block (e.g., a *file* or a *row group*), but this can be abstracted away in a general discussion of the SPA framework. For cloud-native engines, immutable blocks are a necessity, as cloud object stores such as Amazon S3, Google Cloud Storage, or Azure Blob Storage only allow immutable objects in the first place.

**Block-based, partial indexing.** Like traditional *secondary* indexes, SPA only manages additional indexes for speeding up queries and does not affect the primary data representation. But unlike traditional secondary indexes, SPA constructs *partial* indexes at the block granularity, which has two implications: first, only a subset of blocks of a table may be covered by indexes at any given point in time; second, key values are mapped to block identifiers rather than individual tuples. Thus, as in other filtering techniques, the benefit comes from skipping blocks during table scans and saving I/O costs, rather than answering queries using only indexes or optimizing the intra-block scan efficiency. Of course, a particular implementation of SPA could go beyond block skipping, but the complexity and implications of this design choice are beyond the scope of this vision paper. Similarly, potential reorganization or replication of primary storage to optimize predicates [10]—as an alternative to secondary structures—is an orthogonal concern.

Note that we use the term partial to imply not only a *partially built* index, but also *partial coverage* of the tuples in primary storage. Other adaptive indexing approaches like database cracking [11] are partial only in the sense that an index is maintained *partially sorted*, but it still covers all values of the indexed columns. SPA indexes, on the other hand, only cover a subset of blocks of primary storage.

### B. Workload-driven operation

As queries come in, the SPA framework tracks predicates that have been pushed down to table scan operators and selects those that are good candidates for acceleration. The goal is to identify opportunities for indexing, i.e., cases where predicates are selective but the existing filters are ineffective (because too many blocks are scanned without finding a matching tuple). More interestingly, there might be predicates where filters are effective for some blocks but not for others; in this case, the incremental nature of SPA enables finding an optimal combination of filters and indexes.

**Predicate tracking.** Predicates can refer to multiple columns and contain arbitrary expression trees—usually in disjunctive normal form—of which the framework can select one or multiple sub-expressions for tracking and indexing. Furthermore, to account for skew within each predicate, the same expression can be tracked either as a general pattern (e.g., $a < ?$) or with literals instantiated (e.g., $a < 0$). Related work on semantic caching [12], [13] provides a solid foundation for selecting and manipulating predicates, which we consider orthogonal to the framework. For each selected predicate, the framework keeps track of the selectivity and the effectiveness of block skipping observed in scans. The general framework does not dictate how to track and compute these metrics. A simple implementation might record how many times each block is accessed without finding a matching tuple and divide this value by the total number of accesses. A high value then indicates that queries containing that particular predicate might benefit from indexing; in an ideal scenario, this metric would be zero.

**Cost and benefit metrics.** In addition to predicate statistics, the framework keeps track of cost (for both space consumption and maintenance) and benefit metrics for each index structure it maintains. It is useful to think of these metrics in terms of

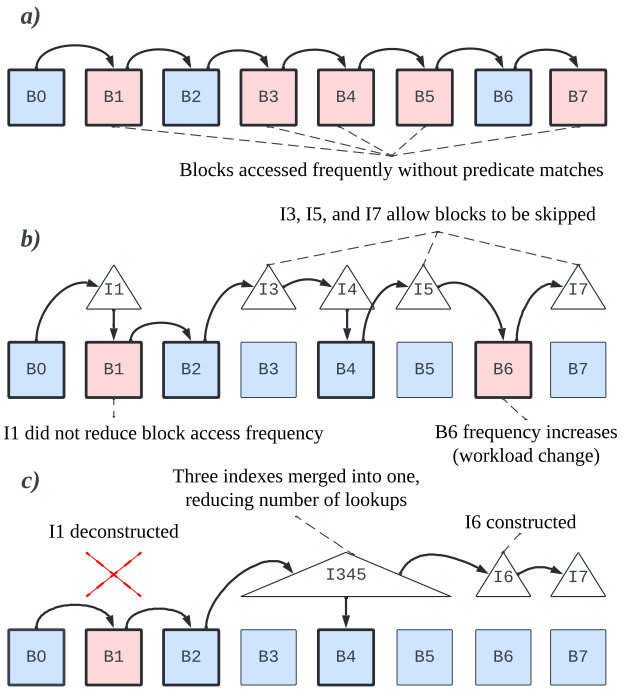


Fig. 1. Dynamic index construction, merge, and deconstruction. (color: access frequency, arrows: example query)

*economics*, i.e., as investment and returns: an index is useful for as long as the cost it incurs justifies the savings it provides in avoiding full table scans. Lastly, statistics on a per-block level are useful to implement policies for deconstructing or evicting indexes under a shifting workload; for instance, if data analysis tends to focus more on the last months, then the economic benefit of indexing will gradually shift away as blocks age, eventually removing their indexes.

**State maintenance.** With costs and benefits being tracked, it might appear that SPA relies on fine-grained maintenance of many metrics, which is a challenge on its own, especially in a distributed system. However, SPA does not prescribe the granularity and accuracy of tracking. A system might opt to track only the most impactful predicates, or it could entirely rely on probabilistic models. For example, one could decide to construct a partial index with a certain probability whenever a block was scanned but had no matches for a predicate. Such a design would incur minimal state maintenance compared to a deterministic and fine-granular approach.

**Example.** To illustrate the dynamic lifecycle of indexes, Figure 1 shows a table consisting of eight blocks (`B0`-`B7`) across three snapshots in time ($a, b, c$), from top to bottom. The blocks are colored based on how frequently they are accessed without a predicate match: red blocks have high frequency, and thus are good candidates for indexing; blue blocks have low frequency, which means they either have matching tuples very often or are skipped very often thanks to an existing index. The arrows represent the blocks and indexes accessed in an arbitrary table scan with the relevant predicate. In snapshot $a$, there are no indexes. Thus, every block is accessed. In snapshot $b$, indexes `I1`, `I3`, `I4`, `I5`, and `I7` are constructed on their corresponding blocks. The scan only skips blocks `B3`, `B5`, and `B7`, as the other indexed blocks (`B1` and `B4`) contain matching tuples. Snapshot $c$ illustrates the merge and deconstruction of indexes, which we discuss in the following.

### C. Smooth, cost-bounded index maintenance

After or during table scans, the collected statistics might indicate that more indexing would benefit a particular predicate. This can be the case if, e.g., the ratio between the number of blocks not containing matches and the number of blocks scanned is above a certain threshold. Once an opportunity for indexing is detected, the cost statistics dictate if the SPA framework should construct indexes or not, i.e., if there is currently enough budget for further indexing. At first, indexes are constructed at the lowest level of granularity, which is a block. The choice of which block to index could be random or guided by per-block statistics. As SPA constructs and deconstructs partial indexes with a fine granularity and never as a whole, the resulting query performance is smooth, meaning that no sudden performance cliffs are to be anticipated.

**Index construction.** Once an index is constructed for a particular block, the cost incurred by that construction is deducted from the available budget. When the table is scanned again with the same predicate, the scan logic detects the existence of an index for that block and probes it. If no matches are found, or all data needed can be retrieved from the index, then the scan operator can skip that block and the index has provided a benefit. Otherwise, the index has not provided any benefit, and in fact the additional cost of such lookups could be detrimental in the long run. Both of these events inform the cost-benefit statistics and also determine the budget available to construct more indexes in the future. This budget is what allows the framework to bound the additional cost of indexing on average, and tweaking the mechanism by which it increases enables a choice between conservative and aggressive strategies.

**Index merge.** As more indexing budget is accrued and indexes on individual blocks prove their benefit, the framework may merge them into larger, multi-block indexes, which are overall more compact and efficient: Larger indexes keep track of a list of blocks for each key value, and they allow the scan operator to skip multiple blocks with a single lookup. Merge operations are guided by and affect the budget in the same way as the initial construction of single-block indexes. When an indexing opportunity is identified in the predicate statistics, the framework can choose between constructing new indexes or merging existing ones. Once again, this choice can be random or guided by additional statistics. The simplest implementation only takes localized decisions (e.g., by only merging indexes on neighboring blocks) rather than global ones. In workloads with many selective queries on the same predicate, SPA will eventually converge into a full index.

**Index deconstruction.** In addition to constructing and merging indexes, the SPA framework also needs a mechanism to deconstruct indexes over time (i.e., drop single-block indexes or split large indexes into single-block ones). There are two main driving forces behind such mechanism, which a cost model can combine in different ways: the storage cost of index structures over time, and the lookup cost (including CPU and I/O cost) of indexes when blocks cannot be skipped. A model that focuses primarily on the storage cost would be a cache, in which case deletes would only take place when free space is low. On the other hand, a model focused on lookup cost could deconstruct already when the utility of an index decreases. The latter model can be especially useful in a cloud context where access cost dominates space cost and storage is elastic and virtually limitless. Indexes that are deemed not worth their cost by the model are deconstructed. There are many possible ways of deciding at what point to deconstruct indexes and SPA does not prescribe a certain way. Even simplistic, stateless strategies like deconstructing random indexes might be feasible, as useful indexes would be eventually re-constructed automatically. Since deconstruction is fundamentally guided by usefulness, an average steady state is achieved in which only useful indexes are kept alive—without bulk or offline maintenance tasks.

**Merge and deconstruction example.** The example of Figure 1c shows how indexes can be deconstructed and merged in a dynamic workload. Index `I1` did not lead to a reduction of `B1`'s access frequency after its construction in snapshot $b$, so it is deconstructed in snapshot $c$. In contrast, because `B6`'s frequency increased from the measurement in snapshot $a$ (i.e., the block became red), a new index `I6` is constructed. Lastly, indexes `I3`, `I4`, and `I5` are merged, allowing blocks `B3` and `B5` to be skipped in a single lookup.

**Data updates.** As we assume immutable blocks and copy-on-write updates, a value in a block is never updated in place but rather deleted and reinserted. Consequently, indexes also are immutable like the blocks they refer to. Deletion of whole blocks can be handled automatically in a deferred way, since indexes on a deleted block gradually lose their value with respect to the cost model and thus become candidates for deconstruction. Similarly, creating a block does not trigger any immediate action in the SPA framework, as index construction is guided by usefulness in table scans. These properties imply that data updates are entirely decoupled from index maintenance. Thus, SPA can be used to accelerate data-lake query engines that do not control data updates and ingestion.

### D. Economic cost modeling

**Indexing cost.** Keeping track of maintenance costs and benefits allows SPA to bound the compute overhead of index maintenance, but it does not answer the question of how much space to use for indexes. Space vs. time is, of course, a classical computer science trade-off, and a simple approach would be to rely on unused storage capacity or let the user decide how much space to invest on secondary indexing. However, we argue that pushing this responsibility to users is unsatisfactory: how should a user know how much space to use? In a cloud setting, where resources can be allocated on demand, one can instead incorporate both compute and storage costs into one model that optimizes overall workload cost.

**Workload cost optimization.** Let us consider an example scenario—based on realistic prices—where keeping 1TB of data in a cloud object store costs $25 per month, and one processor core costs $50 per month. These prices imply that a 2TB index is economically worthwhile as long as it saves more than one CPU-month of work. But of course, construction of that index also has compute and transfer costs, which must be added to the investment part of the investment/return calculation. Assuming a one-time cost of $25 to compute and store a terabyte of index data, a 1TB index would be at the break-even point for the scenario above. A fully cloud-native implementation of SPA would incorporate the pricing model for the cloud in use, and therefore be able to directly optimize for overall workload cost. We believe such an economic model is not only more appropriate for the cloud, but also makes for a simpler implementation of the SPA framework. Note that in a caching-oriented model where space is fixed, the decision of how much space to allocate for indexes is taken *outside* the framework, whereas the proposed approach makes that decision *within* the framework.

### E. Index-structure modularity

Our presentation of SPA so far used the general term *index* without assuming any underlying data structure. A particular implementation may either assume a single type of index, e.g., B-trees, or support multiple types that can be chosen depending on the access pattern. For equality predicates, for instance, the system might choose to construct block-level hash indexes like cuckoo index [14] or—in the approximate case—Bloom filters [15]. The latter are already quite common in column-store systems, but the benefit of SPA is that no upfront decision is required when loading the data, and filters are only constructed where they indeed provide a benefit to the workload. With range predicates, ordered indexes like B-trees or radix trees can be considered.

**Interaction with static filters.** Simple static filters like min/max statistics could in principle be managed by SPA, but given their simplicity and small, bounded cost to both compute and store, a simple and effective implementation would likely always create them on every block. We argue that SPA should manage only more complex structures, e.g., when size depends on the key distribution in each block, when cost to compute at load time is not negligible, or when effectiveness is sensitive to workload characteristics. Bloom filters are an example of such structures. On the other hand, certain types of SPA indexes could aim for a symbiosis with min/max statistics. For example, the partial index could be used to store (all data of) frequently accessed items in the block, yielding a form of partial semantic caching. Alternatively, the index could store outliers explicitly and tighten min/max bounds on the underlying block, making skipping more effective [5].

**Specialization and refinement.** Depending on the types of predicates and various statistics collected by the framework, more specialized structures might be chosen. Similarly, structures can be refined to increase precision at the cost of additional space. For instance, a SuRF data structure [16] can be incrementally rebuilt to eventually form an adaptive radix tree (ART) [17] and cover an entire key space. For arbitrary predicates where a particular constant is used very frequently (e.g., $a > 0$), compressed bitmaps (where a bit is set for tuples where the predicate matches) can also be very effective—these can be seen as a predicate evaluation cache, used in systems like PowerDrill [18].

**Mergeability.** One important property of index structures that SPA considers is *mergeability*, i.e., the ability to construct large indexes from multiple smaller indexes on subsets of the data. In principle, indexes can always be reconstructed on the large sets by re-scanning the primary data, but it is desirable to construct large indexes by simply merging existing ones.

## III. Proof of concept

SPA is a general framework—or rather a set of ideas—which we introduced in the previous sections and from which multiple design instantiations are possible. In this section, we discuss a bare-bones algorithm for such an instantiation and discuss a proof-of-concept implementation and evaluation in the Hyper database system [19]. We scan data stored in the Apache Parquet file format [1]. The blocks on which SPA operates are the row groups in the Parquet file. We use a simple cost model tailored to index maintenance in main memory.

### A. Index construction

Whenever a predicate $p_c$ on a column $c$ is pushed down to a scan operator, an index on that column might be helpful. Therefore, we maintain an indexing budget $b_c$ for each column. We need the following parameters for our model:

- Deposit factor $f_d$: How much additional query time are we willing to spend to construct indexes? For example, with $f_d = 0.1$, 10% more time per query can be spent on index construction.
- Reinvest factor $f_r$: What fraction of time saved by using an index do we want to reinvest into constructing further indexes? For example, with $f_r = 0.5$, 50% of the time saved by using indexes can be re-invested to construct further indexes on the column.

Our access method is a table scan with block skipping: Scanning a table with a predicate $p_c$ on a column $c$ requires scanning or skipping each block of that table. For each block, we first do a cheap skipping check using the min/max bounds of the row group stored in the Parquet file. If this quick check does not allow us to skip the block, we check whether there is an index on $c$ for this block.

If there is no index on $c$ for this block yet, we scan the block, taking time $t_{\text{scan}}$. We apply $p_c$ and check whether it filtered out all tuples in the block. If it did, an index on $c$ for this block would have been helpful. We add $f_d \cdot t_{\text{scan}}$ (the fraction we are willing to invest) to our indexing budget $b_c$. Then, given that constructing an index on the block would take time $t_{\text{build}}$, we check whether we have accrued enough budget (i.e., $b_c \geq t_{\text{build}}$). If we have, we construct an index on this block and deduct $t_{\text{build}}$ from $b_c$. Construction is performed directly by the thread(s) executing the query. The I/O cost of index construction is low, as the values are already loaded.

If there is already an index on $c$ for the block we are scanning, we probe that index to see whether $p_c$ can have a match. Probing the index takes time $t_{\text{probe}}$ and we assume that $t_{\text{probe}} \ll t_{\text{scan}}$. If we probe an index for $p_c$ and it returns that $p_c$ could match, we deduct a penalty of $t_{\text{probe}}$ from our indexing time budget $b_c$, as the index was *not* helpful in this case, as we need to scan the block anyway. This penalty prevents us from retaining indexes for non-selective predicates. In case the index probe returns that this block cannot have a match, we do not need to scan it and therefore saved $t_{\text{saved}} = t_{\text{scan}} - t_{\text{probe}}$. We reinvest the fraction $f_r$ of the saved time and therefore add $f_r \cdot t_{\text{saved}}$ to our budget $b_c$.

### B. Index deconstruction

For our proof-of-concept, we pick the most simple yet effective strategy of simply deconstructing indexes at random whenever we run into memory pressure. The adaptive construction process ensures that high-value indexes will quickly be re-constructed. The advantage of this approach is that it does not require to keep any state about the usefulness of indexes besides $b_c$.

### C. Algorithm properties

Even though this algorithm is as simple as it gets, it displays the following desirable properties:

- The additional processing time spent indexing is roughly bounded by $f_d$, so no query will be significantly slower if this value is chosen low enough.
- We will never index a block that contains matches for all predicates $p_c$ we see on column $c$. Thus, in case we have skewed data where some blocks always contain matches while others contain no matches, we will only index the blocks where the index can indeed help us skipping.
- The more selective a predicate is, the more blocks will contain no matches, the faster we will accrue budget, and the faster we will construct an index.
- In case a column often has unselective predicates, an index that does not help with skipping these predicates could become a burden. By penalizing probing an index in vain with $t_{\text{probe}}$, we dampen the index construction on the column, again leading to a state where columns with more selective predicates have more indexes constructed than ones with less selective ones.
- The more queries with a selective predicate on $p_c$ are issued, the faster the index on $c$ will be constructed, so we favor columns that are queried and restricted often.

### D. Evaluation

We implemented the algorithm discussed above in the Hyper database system scanning Apache Parquet files. In our

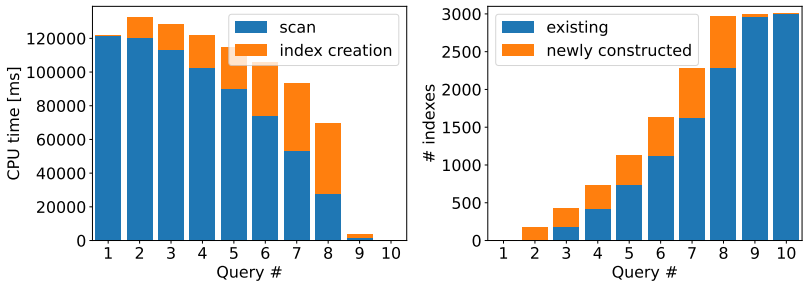


Fig. 2. CPU time (left) and number of constructed indexes (right) when running queries with $f_r = 0.5$ & $f_d = 0.1$

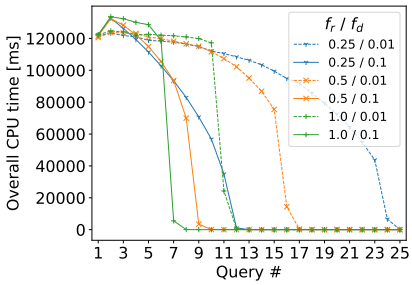


Fig. 3. Overall CPU time when running queries with varying $f_r$ & $f_d$

experiments, we keep the adaptively-built indexes in main memory. We opt for a simple index: A sorted list of the domain values. Binary search on this sorted list is used to check whether a value is in the block. We could also answer range queries with this index structure, but we focus on equality predicates here. For further simplicity, we do not merge these lists but only build single-block lists.

Our focus is on selective predicates where simple min/max pruning is not sufficient, as these are the only predicates where SPA will construct an adaptive index. One example for such queries are look-ups of a specific UUID. UUIDs are often used as key or foreign-key columns and as UUIDs are pseudo-random, their value is usually not clustered or correlated to the insertion time, so min/max pruning does not help. We scan a 22GB Parquet file containing 600 million rows in 3000 row groups. The file mimics the TPC-H scale factor 100 lineitem table with an added unique UUID column. The test is run on an AWS EC2 instance of type m5d.8xlarge. All measured times are CPU time. We run the following query repeatedly:

```
SELECT * FROM 'lineitem.parquet'
WHERE uuid = '<random UUID present in the table>'
```

As the values are unique, only one of the 3000 row groups in the file ever contains a matching tuple. Therefore, all row groups are viable candidates for index construction. Figure 2 shows the results after running the query 10 times. On the first run, adaptive indexing is disabled. After the first query, we enable adaptive indexing using the parameters $f_r = 0.5$ and $f_d = 0.1$. Thus, we are willing to invest an extra 10% of time into index construction and we use 50% of the time we save by using an index into constructing more indices.

As shown in the figures, we need 8 queries to construct indexes on all 3000 row groups with these settings, so we actually converge rather quickly. Once all blocks have an index, the query is almost four orders of magnitude faster (from 120 seconds CPU time to 20 milliseconds), so our investment quickly pays off. The first query that starts indexing (Query 2) takes indeed around 10% more CPU time than the query that does not index (from 121 seconds to 132 s). It uses this time to construct 182 indexes. In the next query (Query 3), the existing indexes already provide some savings, so that more indexes can be constructed (242), and so on. With Query 9, the time drops drastically, as the previous query constructed almost all remaining indexes, skipping 2966 blocks.

Figure 3 depicts the impact of varying the parameters $f_r$ and $f_d$. We vary $f_r$ between 25%, 50%, and 100%. With $f_r = 100\%$, we re-invest all time we save by using an index into index construction. Therefore, the query time barely gets faster in the first runs, as index construction eats up all time saved. However, as it uses all time to construct indexes, it finishes constructing all indexes the fastest and then the query time quickly drops. With $f_r = 25\%$, we only invest one fourth of the savings back into index construction, so we converge slower but smoother. $f_d$ is varied between 10% and 1%, which are both conservative, barely noticeable slow-downs. Even with these conservative values, the algorithm converges quickly building indexes on all blocks of the table. Our vision for SPA is that the initial deposit should not hurt query latency visibly, so we advise against being more aggressive here.

This proof-of-concept experiment shows the feasibility of constructing an index adaptively during a scan and guided by a cost model. The experiment is simplified, as we only construct the indexes in main memory, so we converge exceptionally quickly. For the sake of brevity, we neither did incorporate an experiment using cloud storage nor an experiment where index deconstruction or data updates are relevant. An index stored on cloud storage will be more costly to construct and maintain, so the approach will converge slower. $f_d$ might be chosen slightly higher to counteract this. All such experiments are good candidates for future research.

## IV. Research Agenda

**Related research themes.** The SPA framework is intended to spur research into a number of areas, emboldened by our initial results and the observation that *economical* principles like deposits, gains, and re-investment can be unified with data management—with monetary workload cost becoming the primary optimization goal [20]. The virtually limitless resources of public clouds and their known performance/price-points and quantifiable economic cost can break the barriers to adoption that have historically stood in the way of, e.g., completely automatic physical design [21]. In other words, quantifiable value improvements hold promise for completely automatic decision-making. This realization we marry with well-known design principles that aim for performance robustness [22], smoothness [23], and adaptive indexing [11].

**Economic decision making.** We call for cross-disciplinary work between economics, econometrics, and data management to develop economic strategies that guide data management decision making. While in this paper we spoke of a single economic actor, in cloud environments the cloud provider is another actor who could make smooth investment decisions to offer value propositions to their users. In the future, the optimization of workloads could become the terrain of investment-willing third parties.

**Economic data structures.** We mention index structures such as bloom and cuckoo filters that can exclude the possibilities of matching. These data structures are probabilistic in the sense that there is a false-hit probability and we usually attempt to minimize this. Rather than optimizing a false-hit probability or performance optimality [15], SPA data structures should maximize economic value.

**Partial indexing and caching.** An important observation is that workloads make certain data much more valuable to index than other, and this leads to opportunities for indexes that are by definition partial. While most indexing examples in this vision paper aim for negative queries that trigger the skipping of a block, one can also design data structures that benefit from positive queries (semantic caching [13], [12]).

**Data reorganization.** In addition to min/max pruning, some systems offer user-defined [24] or even automatic [25] data clustering. This can be considered a kind of clustered index—but only one clustering per table can be specified. In this paper, we assumed that the primary data representation is not changed by SPA. However, the SPA framework could construct and maintain several physical data representations—rather than just block-based indexes.

**State management.** As discussed, SPA does not prescribe how to keep track of cost and benefit metrics and we proposed some simple approaches. The trade-offs of a model tracking more state versus a low-maintenance probabilistic model might be a promising field for future research.

**Query processing.** The SPA framework calls for innovation in areas like robust query processing, as there are many opportunities to reduce performance cliffs in query processing still, as well as query optimization, taking into account the presence of the SPA framework in query optimization decisions.

## ACKNOWLEDGEMENTS

This paper is an outcome of Dagstuhl Seminar 22111 on Database Indexing and Query Processing. We thank the organizers (Goetz Graefe, Renata Borovica-Gajic, Allison Lee, Pinar Tözün) and other participants for their feedback.

## REFERENCES

[1] "Apache parquet," https://parquet.apache.org/, 2022.

[2] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper, "Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation," in *SIGMOD*. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2882903.2882925

[3] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner, "Hyrise re-engineered: An extensible database system for research in relational in-memory data management," in *Proc. EDBT*, 2019. [Online]. Available: https://doi.org/10.5441/002/edbt.2019.28

[4] G. Moerkotte, "Small materialized aggregates: A light weight index structure for data warehousing," in *VLDB*, 1998. [Online]. Available: http://www.vldb.org/conf/1998/p476.pdf

[5] G. Graefe, "Fast loads and fast queries," in *DaWaK*, 2009. [Online]. Available: https://doi.org/10.1007/978-3-642-03730-6_10

[6] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *SIGMOD*. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2882903.2903741

[7] "Apache ORC," https://orc.apache.org/, 2022.

[8] M. Armbrust, T. Das, S. Paranjpye, R. Xin, S. Zhu, A. Ghodsi, B. Yavuz, M. Murthy, J. Torres, L. Sun, P. A. Boncz, M. Mokhtar, H. V. Hovell, A. Ionescu, A. Luszczak, M. Switakowski, T. Ueshin, X. Li, M. Szafranski, P. Senster, and M. Zaharia, "Delta lake: High-performance ACID table storage over cloud object stores," *PVLDB*, vol. 13, no. 12, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3411-armbrust.pdf

[9] "Apache iceberg," https://iceberg.apache.org/, 2022.

[10] L. Sun, M. J. Franklin, J. Wang, and E. Wu, "Skipping-oriented partitioning for columnar layouts," *PVLDB*, vol. 10, no. 4, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol10/p421-sun.pdf

[11] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe, "Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores," *PVLDB*, vol. 4, no. 9, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p586-idreos.pdf

[12] D. Durner, B. Chandramouli, and Y. Li, "Crystal: A unified cache storage system for analytical databases," *PVLDB*, vol. 14, no. 11, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p2432-durner.pdf

[13] S. Dar, M. J. Franklin, B. Þ. Jónsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *VLDB*, 1996. [Online]. Available: http://www.vldb.org/conf/1996/P330.PDF

[14] A. Kipf, D. Chromejko, A. Hall, P. A. Boncz, and D. G. Andersen, "Cuckoo index: A lightweight secondary index structure," *PVLDB*, vol. 13, no. 13, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3559-kipf.pdf

[15] H. Lang, T. Neumann, A. Kemper, and P. A. Boncz, "Performance-optimal filtering: Bloom overtakes cuckoo at high-throughput," *PVLDB*, vol. 12, no. 5, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p502-lang.pdf

[16] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *SIGMOD*, 2018. [Online]. Available: https://doi.org/10.1145/3183713.3196931

[17] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*, 2013. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544812

[18] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser, "Processing a trillion cells per mouse click," *PVLDB*, vol. 5, no. 11, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1436_alexanderhall_vldb2012.pdf

[19] "Hyper API," https://www.tableau.com/developer/tools/hyper-api, 2022.

[20] V. Leis and M. Kuschewski, "Towards cost-optimal query processing in the cloud," *PVLDB*, vol. 14, no. 9, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1606-leis.pdf

[21] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *VLDB*, 2000. [Online]. Available: http://www.vldb.org/conf/2000/P496.pdf

[22] G. Graefe, H. A. Kuno, and J. L. Wiener, "Visualizing the robustness of query execution," in *CIDR*, 2009. [Online]. Available: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_82.pdf

[23] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser, "Smooth scan: robust access path selection without cardinality estimation," *VLDB J.*, vol. 27, no. 4, 2018. [Online]. Available: https://doi.org/10.1007/s00778-018-0507-8

[24] "Micro-partitions & data clustering - snowflake documentation," https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions.html, 2022.

[25] "Automatic clustering - snowflake documentation," https://docs.snowflake.com/en/user-guide/tables-auto-reclustering.html, 2022.