SmartRabbit: An Interactive Query Processor

PRATYOY DAS, University of California, Irvine, USA
MARTIN BOISSIER, Hasso Plattner Institute, Germany
KYOUNGMIN KIM, EPFL, Switzerland
SHARAD MEHROTRA, University of California, Irvine, USA
TILMANN RABL, Hasso Plattner Institute, Germany

Traditional relational database systems optimize analytical queries to minimize their end-to-end latency. The resulting optimal plans are usually *blocking*, forcing users to wait until full query completion before seeing any results. This execution model precludes *interactivity*, i.e., users cannot observe partial results or gain early insights for long-running queries. Query optimizers rarely choose plans that promote interactivity, since such plans either incur prohibitively large latencies or involve operators for which interactive alternatives are often infeasible. This paper introduces a novel interactive query processor named SmartRabbit that promotes interactivity of answers while matching the end-to-end latency of blocking execution plans. We achieve this by first designing a plan optimized for interactivity for a given query, and then simultaneously executing this plan alongside a traditional blocking plan. The two executions are carefully synchronized to maintain the correct order of answers and prevent duplicates. We implement SmartRabbit in a scalable, open-source database system and show that SmartRabbit consistently delivers early and continuous results across various analytical benchmarks, data scales, and levels of parallelism, with only marginal latency overhead compared to traditional blocking execution.

CCS Concepts: • **Information systems → Relational database model**; *Physical data models*; **Triggers and rules**; **DBMS engine architectures**; **Relational parallel and distributed DBMSs**; **Online analytical processing engines**.

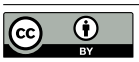
Additional Key Words and Phrases: Interactivity, Query Processing, Query Optimization

ACM Reference Format:
Pratyoy Das, Martin Boissier, Kyoungmin Kim, Sharad Mehrotra, and Tilmann Rabl. 2026. SmartRabbit: An Interactive Query Processor. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 235 (June 2026), 25 pages. https://doi.org/10.1145/3802112

1 Introduction

Relational database systems typically optimize and process large analytical queries with the objective of minimizing end-to-end latencies, i.e., the time between arrival of the query to returning the full answer set [15, 40]. Resulting execution plans are typically blocking, whether for systems using an operator-at-a-time (e.g., MonetDB [19], Hyrise [12]) or a pipelining model (e.g., DuckDB [37], Umbra [33]). While pipelining models continuously pipe intermediate results through operators, operations such as sorting or aggregations are typically blocking (so-called *pipeline breakers*),

Authors' Contact Information: Pratyoy Das, pratyoyd@uci.edu, Department of Computer Science, University of California, Irvine, Irvine, USA; Martin Boissier, martin.boissier@hpi.de, Hasso Plattner Institute, Potsdam, Germany; Kyoungmin Kim, kyoung-min.kim@epfl.ch, EPFL, Lausanne, Switzerland; Sharad Mehrotra, sharad@ics.uci.edu, Department of Computer Science, University of California, Irvine, Irvine, USA; Tilmann Rabl, Tilmann.Rabl@hpi.de, Hasso Plattner Institute, Potsdam, Germany.



ACM 2836-6573/2026/6-ART235
https://doi.org/10.1145/3802112

leading to results being returned only at the end of the blocking phase when the entire input has been processed.

This behavior is undesirable for very large data sets where queries are often running for minutes, as this unnecessarily starves downstream applications who have to sit idle until all the answers arrive at the end. The ability to observe incremental answers during such long-running analytical queries is important to guide further exploration [17]. Analysts also often engage in multistage, hypothesis-driven workflows in which the outcome of one query determines the next [13]. In such settings, early and continuous result delivery shortens the feedback loop and enables faster decision-making. This motivates query executions that can expose results incrementally rather than only upon completion.

A concrete motivation for such query processing today arises from the growing use of *agent-based* systems for data analysis. In emerging commercial platforms (e.g., Microsoft Fabric Copilot [27, 28], Databricks' "Agent Bricks" [10], and Snowflake's agent frameworks [39]), users specify high-level analytical goals, while the system automatically generates and executes sequences of exploratory SQL queries. In such workflows, analytical decisions can often be made based on *early results* from intermediate queries, rather than complete outputs.

Consider an analyst investigating a sudden drop in heater sales during a particular week in the U.S. Northeast. Several explanations are plausible, including a website outage, unusual weather patterns, inventory shortages, delayed shipments, or competitor promotions. Each hypothesis triggers one or more analytical queries over large datasets (e.g., historical sales, clickstream logs, inventory tables, logistics records, or external weather feeds). Whether the analyst (or an AI agent acting on their behalf) continues along a given analytical path often depends on the *first few result tuples*. For example, if early results show that the decline is localized to a small set of zip codes, the analyst may pivot to a logistics investigation; if the effect is uniform across the region, weather-related explanations become more likely. Waiting minutes for full query completion significantly slows this decision loop, whereas early answers allow many analytical paths to be abandoned quickly.

These scenarios impose two key requirements on the underlying query engine. First, it must return *early, correct partial results* so that analytical paths can be quickly pursued or pruned. Second, it must operate effectively over very large-scale datasets.

Prior work has explored several ways to expose answers early, including sampling-based approximate query processing (AQP) [1, 35], `LIMIT`/top-k processing [8, 20], and physical-plan choices that favor early tuple production, such as preferring nested-loop or index nested-loop joins over hash joins. However, these approaches have important limitations. AQP targets approximate answers, typically for aggregation-oriented analytical queries, rather than early exact answers for general SQL workloads. The `LIMIT` clause allows engines to terminate execution early, which optimizers often exploit via limit pushdown. Limits generally cannot be pushed below blocking operators (pipeline breakers) such as hash-based `GROUP BY` or unindexed `ORDER BY` though, since these operators must fully materialize their inputs before producing even the first output tuple. Finally, physical operators that favor early tuple production are often substantially slower than latency-optimized alternatives.

We therefore target query execution that returns early, exact results while still completing the full query efficiently. Moreover, the results produced so far must be *prefix-correct*: at any point during execution, they must form a valid prefix of the final query output under a specific output order, i.e., no tuple that should appear earlier is produced later, and no duplicates are emitted. We refer to execution with these properties as *interactive*, and to a plan that enables it as an *interactive plan*.

In this paper, we introduce SmartRabbit, a novel dual-execution query processor for interactive query processing. SmartRabbit executes each query using two coordinated plans: an *interactive* plan that produces early, correct results, and a latency-optimized *blocking plan* that runs in the background to efficiently complete the query. SmartRabbit does not introduce new physical operators; instead, it constructs interactive plans using well-known physical operator techniques, as summarized in Section 2.

We formalize the goal of SmartRabbit in terms of dual-plan execution below.

Given an SQL query Q, a database instance D, and a baseline physical plan P_b optimized for end-to-end latency, construct and execute a pair of plans (P_i, P_b) such that:

- P_i is an interactive plan that emits prefix-correct result tuples incrementally,
- the combined execution emits no duplicate tuples and preserves query semantics,
- the time-to-first-result (TTFR) is minimized,
- the end-to-end completion time remains comparable to the time taken to execute P_b alone.

As seen in Figure 1, SmartRabbit's key idea is to execute P_i and P_b simultaneously: P_i emits results eagerly to provide early feedback, while P_b progresses toward full query completion. Once P_b is ready to produce results, SmartRabbit halts the interactive execution and seamlessly switches to the blocking plan to produce the remaining results.

Challenges. Producing early, exact query results while preserving the efficiency of a latency-optimized plan raises several challenges. First, results must satisfy query semantics (e.g., `ORDER BY`) without being provisional. Second, the system requires valid interactive plan construction and non-blocking implementations of operators such as `GROUP BY`. Third, the interactive and blocking executions must be coordinated to avoid duplicate tuples during handoff. Finally, dual execution must scale in distributed settings with minimal overhead, so that the cost of interactivity does not outweigh the benefit of early results.

We implemented SmartRabbit within Apache AsterixDB [3], an open-source, cluster-scale analytical database with native support for secondary indexes, which are central to interactive execution. AsterixDB separates query compilation and optimization from distributed execution through its compiler and Hyracks-based runtime [8], enabling new physical operators, rewrite rules, and execution behaviors to be introduced in a localized manner. This makes it well suited for realizing and evaluating SmartRabbit's hybrid execution strategy. Experimental results show that SmartRabbit produces interactive results across multiple benchmarks, query scales, and degrees of database parallelism, with little or no additional latency relative to conventional blocking plans.

Costs and applicability. Enabling interactive execution in a query engine comes with trade-offs: SmartRabbit may require additional secondary indexes to facilitate the interactive plan (which are listed in Table 2 for our experiments), and the storage and maintenance of these indexes are an overhead. It also introduces a small overhead in the blocking execution due to coordination with the interactive execution. Further, SmartRabbit specifically targets queries whose subqueries can be

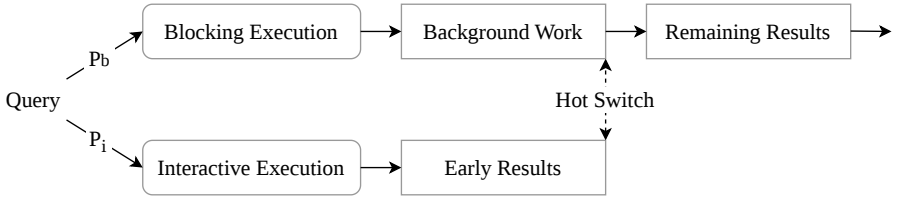


Fig. 1. Key idea of SmartRabbit

decorrelated by the logical optimizer and that do not require ungrouped global aggregation. Across TPC-H, SSB, and ClickBench, we find that approximately 70% of queries satisfy these criteria.

Contributions. Our novelty lies in the dual-execution query answering model and the coordination mechanisms that guarantee prefix-correct, duplicate-free output while our plan construction and operator choices build on well known physical planning and operator implementations. The key contributions of this work are

- We introduce SmartRabbit, a novel dual-execution query processing strategy. SmartRabbit executes an optimized interactive plan alongside a blocking plan to deliver prefix-correct partial answers incrementally while ensuring end-to-end latency of optimized blocking plans used in existing databases.
- We develop a theoretical framework to formalize when a physical operator is "interactive" with respect to its inputs (Section 2), and study different physical implementations of common logical operators under the framework.
- We provide a rule-based framework, inspired by interesting-order planning [40], that constructs interactive plans compatible with the corresponding blocking plans. (Section 3)
- We implement the core mechanisms needed to implement SmartRabbit to run on AsterixDB. We also provide implementation details for traditionally blocking operations needed to emit early exact results under the model. (Section 4)
- We evaluate SmartRabbit on TPC-H, SSB, and ClickBench across scales, quantifying TTFR/interactivity gains, overheads, and applicability at large data scales. (Section 5)

We believe this work spotlights the promise of Interactive Query Processing and facilitates follow-on work on plan and execution design, handoff policies, and interactivity-aware cost models.

2 Interactive Operators and Plans

We first define interactive query execution as:

Definition 2.1 (Interactive Query Execution). Given a query Q with result set $R(Q)$, an execution is interactive if it produces correct result tuples incrementally, such that a non-empty prefix of $R(Q)$ is emitted strictly before the completion time of a latency-optimized (blocking) execution of Q.

Unlike classical pipelining, which characterizes *how tuples flow between physical operators*, interactivity characterizes *when correct end results become observable externally*. Interactivity is, therefore, a user-facing semantic property, wherein the system generates and delivers results to the user iteratively. A pipelined execution architecture is necessary for interactivity, but not sufficient since often query plans contain pipeline breakers (e.g., sorting, hashing) that require upstream operators to finish execution and deliver all the results before the operator can produce any results.

In order to understand how to answer a query interactively, we need to understand how to design each operator of the query interactively. Some operators [1] , such as index scans, can produce results almost immediately, while others, like aggregation without grouping, require all input tuples before yielding any output. To capture this spectrum, we define two temporal markers for every operator: the moment it can first emit an output, and the moment it is guaranteed to have produced all outputs. We refer to these as the *inception point* and *completion point* of the operator respectively.

[1]Terminology. In the remainder of this section, we use the term *operator* to mean a physical operator not a logical relational operator.

2.1 Inception Point and Completion Point

Definition 2.2 (Inception Point). The *inception point* (ip) of an operator is the earliest time it can produce any output after its first input arrives.

Definition 2.3 (Completion Point). The *completion point* (cp) of an operator is the earliest time at which the operator has produced all of its output tuples after its first input arrives.

While inception and completion points characterize the interactive potential of individual operators, query execution involves multiple operators typically connected in a directed acyclic graph [15]. Let p and c denote two operators such that c consumes the output of p; we refer to p as the *producer* (or upstream) operator and c as the *child* (or downstream) operator. An operator's ability to produce results thus depends not only on its own behavior but also on when its inputs become available from its producer(s). This motivates a notion of *blocking* between operators, which captures how dependencies propagate interactivity (or lack thereof) across the plan. We now present the following definitions and lemmas.

Definition 2.4 (Blocking Operator). For an edge $p \rightarrow c$ where c consumes the output of p, we say that c is *blocking on* p if c cannot produce any output until p has fully completed.

Lemma 2.1 (Blocking Characterization). *For an edge $p \rightarrow c$ between operators, c is blocking on p if and only if*

$$\mathrm{ip}_c \;\geq\; \mathrm{cp}_p \;.$$

Lemma 2.2 (Interactive Characterization). *For an edge $p \rightarrow c$, c is interactive on p if and only if*

$$\mathrm{ip}_c \;<\; \mathrm{cp}_p \;.$$

Definition 2.5 (Interactive Operator Implementation). An operator implementation with multiple input operators $\{p_1, p_2, \ldots, p_k\}$ is *interactive* if it is interactive on every input operator, i.e.,

$$\mathrm{ip}_c \;<\; \mathrm{cp}_{p_i} \qquad \text{for all } i = 1, \ldots, k.$$

Thus, any operator implementation that can start producing outputs without waiting for any of its input operators to complete qualifies as an *interactive operator*. We use this principle to distinguish interactive and blocking implementations of different operators.

2.2 Inception Point and Completion Point of Different Operators

2.2.1 Selection and Projection. Selection and projection are inherently interactive because no implementation of them requires full knowledge of the input before producing output. In a system that pipelines its operators, a selection only evaluates a local predicate on each incoming tuple (or vector of tuples in a block-oriented engine) and can forward qualifying tuples immediately. Similarly, a projection simply rewrites or discards attributes, again producing tuple- or block-at-a-time, without waiting for the entire input. Consequently, under any physical implementation in pipelining systems - these operators produce their first outputs strictly before the producer's completion point, and are therefore interactive.

2.2.2 Joins. The inception point of a join operator is the first time any join result can be produced, and its completion point is when all join results have been produced. An interactive join operator implementation is one which does not require *any* of its joining relations to produce all their tuples before it can produce a join result. Let us look at the different types of join implementations. In this subsection, R and S denote the left and right inputs of a join operator, respectively.

Nested-loop join (NLJ). In a nested loop join, we take a tuple (or a block) of the outer relation (R), and scan the full inner relation (S). A nested loop join operator is potentially ready to produce its first output after it has scanned a tuple (or a block) of the outer and the first tuple from the inner (whether it finds a valid join result is a separate issue). Hence Nested Loop Join is an interactive implementation as it is not blocking on either of its inputs and its inception point can be as low as the maximum of the inception points of both the joining relations:

$$\mathrm{ip}_{\mathrm{NLJ}} \;\geq\; \max(\mathrm{ip}_R, \mathrm{ip}_S).$$

Index nested-loop join (INLJ). In an INLJ, we take a tuple (or a block) of the outer relation and probe an index on the join key of the inner. Since the index is immediately available, the join can produce results as soon as the outer yields its first tuple. Thus the inception point is bounded by the outer relation:

$$\mathrm{ip}_{\mathrm{INLJ}} \;\geq\; \mathrm{ip}_R\,.$$

Because the index provides direct access to matching tuples in S without scanning it entirely, INLJ can produce qualifying join results earlier than a block-oriented NLJ.

Hash join (build on R). In the classical build–probe variant of a hash join, the join must first consume the entire build input R to construct the hash table. Thus it is blocking on R, with

$$\mathrm{ip}_{\mathrm{HJ}} \;\geq\; \mathrm{cp}_R,$$

and no output can be produced until R is complete. By contrast, *hybrid hash join* [15] (HHJ) keeps one partition of R (say R_0) in memory while writing the others to disk. Once R_0 is fully materialized, the join can begin producing results for tuples from L that hash into the same partition. Formally, its inception point can be as low as

$$\mathrm{ip}_{\mathrm{HHJ}} \;\geq\; \max(\mathrm{ip}_S,\, \mathrm{cp}_{R_0}),$$

with $\mathrm{cp}_{R_0} \leq \mathrm{cp}_R$. Hence HHJ may emit results before the entirety of R is read, but typically later than NLJ or INLJ. However it cannot be relied to preserve an existing interesting order as a semantic guarantee.

Sort-merge join (SMJ). In the typical implementation, SMJ first sorts both inputs on the join key and then performs a merge phase to generate join results. Because the sort phase is blocking, the join cannot produce any output until both inputs are fully sorted. Thus SMJ is blocking on *both inputs R and S*, with

$$\mathrm{ip}_{\mathrm{SMJ}} \;\geq\; \max(\mathrm{cp}_R, \mathrm{cp}_S).$$

Progressive/Overlapped joins in literature (Ripple Join, XJoin, HMJ, Progressive Merge Join). These explicitly reduce blocking by overlapping build/probe or interleaving scans, ensuring $\mathrm{ip}_{\mathrm{join}} < \mathrm{cp}_R$ and $\mathrm{ip}_{\mathrm{join}} < \mathrm{cp}_S$ under typical conditions, i.e., earlier inception point than classical hash or sort-merge join [11, 16, 29, 43].

2.2.3 Aggregation. Aggregation operators combine multiple input tuples into one or more result tuples. For a given group g, the group's *inception point* ip_g is the time its first member arrives and its *completion point* cp_g is when the last member arrives from the upstream operator. At the *operator level*, the inception point $\mathrm{ip}_{\mathrm{agg}}$ denotes the earliest time any group g can emit its aggregate.

Without GROUP BY. If no `GROUP BY` is present, aggregation is global, and the operator's inception point is bounded by the completion point of its upstream operator p.

$$\mathrm{ip}_{\mathrm{agg}} \;\geq\; \mathrm{cp}_{\mathrm{p}},$$

so global aggregation is inherently blocking.

With GROUP BY. When grouping is present, each group g has its own cp_g, and the operator can emit results as soon as a group completes. In principle, this allows interactive execution, but the most common implementations are blocking:

Sort-based aggregation. Tuples are sorted on the grouping key and then scanned to produce group aggregates. Because the sort phase is blocking, the operator cannot produce any output until sorting completes. Hence sort-based aggregation is blocking, with

$$\mathrm{ip}_{\mathrm{agg}} \;\geq\; \mathrm{cp}_{\mathrm{sort}}\,.$$

Hash-based aggregation. Tuples are inserted into buckets keyed by the grouping attribute. While partial aggregates can be updated incrementally, the operator cannot guarantee group completion until all input tuples have been processed. Thus hash-based aggregation is blocking, with

$$\mathrm{ip}_{\mathrm{agg}} \;\geq\; \mathrm{cp}_{\mathrm{p}}\,.$$

We discuss two non-blocking aggregation variants below.

Index-based grouping. If an index exists on the grouping attribute, scanning the index yields tuples in group order. Once all tuples for a key have been seen, its aggregate can be emitted immediately. Hence, index-based grouping is interactive, with

$$\mathrm{ip}_{\mathrm{agg}} = \min_{g} \mathrm{cp}_g \;<\; \mathrm{cp}_{\mathrm{p}}\,.$$

This idea of index-driven grouping was first introduced in [40] and leveraged in [17]. Note that even in the absence of an explicit index on the grouping key(s), if the incoming tuples appear in a *sorted order*, we can still perform interactive grouping with the same principle: once we see all tuples with the same grouping key, and the grouping key tuple changes, we are ready to output the aggregate value for that group.

Shuffle on grouping key. A *shuffle* (or *exchange*) operator [14] is commonly used in parallel query engines to redistribute tuples across workers based on a partitioning key. We can consider shuffling as hashing on the grouping attribute, ensuring all tuples of a group reach the same worker. Each worker can then pipeline local aggregation and emit group results once its partition is complete. Hence shuffle-based grouping is interactive, though it may be constrained by available cores and group metadata. In addition, the initial exchange can be a dominating cost factor.

2.2.4 `ORDER BY`. An `ORDER BY` clause is inherently blocking: producing the final sorted output requires reading all input tuples. Formally, its inception point is bounded by the completion of its producer p, i.e.,

$$\mathrm{ip}_{\mathrm{order}} \;\geq\; \mathrm{cp}_{\mathrm{p}}\,.$$

However, `ORDER BY` can be interactive when tuples arrive sorted on the desired attribute from the upstream operator. This occurs, for example, with index scans or with grouped aggregations that preserve order on the grouping attribute. In these cases, results can be streamed immediately while maintaining the global sort order, making the operator interactive i.e.

$$\mathrm{ip}_{\mathrm{order}} \;\geq\; \mathrm{ip}_{\mathrm{p}}\,.$$

By contrast, queries requiring sorting on derived attributes - whose values or orderings are not known a priori - cannot currently be evaluated interactively.

2.2.5 Duplicate Elimination. Duplicate elimination is typically implemented through blocking approaches like sorting or hashing [15]. However, it can be made interactive if the system maintains a data structure (e.g., a hash set) to record tuples that have already been seen; each new tuple can then be checked for membership and emitted immediately if it is distinct. Also, if the input is ordered such that all occurrences of a value appear contiguously – for instance, through an index

scan or a shuffle on the distinct key – each distinct value can be produced as soon as its group is complete. In both cases, the operator's inception point precedes the completion of its input, given by

$$\mathrm{ip}_{\text{distinct}} \;\geq\; \min_{g} \mathrm{cp}_{g} \;<\; \mathrm{cp}_{p} \;.$$

where g ranges over groups of tuples sharing the same distinct key. Distinct-qualified aggregates such as `SUM(DISTINCT)` follow the same ip and cp characteristics as normal aggregates discussed in Section 2.2.3.

From Operators to Plans. The preceding analysis is at the operator level: each operator can be classified by whether its inception precedes its inputs' completion. In SmartRabbit, we focus on creating interactive plans that are made entirely of interactive operators. This allows us to maintain prefix correctness easily. It also allows us to easily define a plan level invariant, whose order is maintained throughout the plan. The next section discusses generating interactive plans in more detail.

3 Generating Interactive Plans

This section explains how to construct an interactive execution plan. We introduce the notion of a fulcrum. We then explain why we always construct interactive plans as left-deep tree plans. We then craft a set of rules that allow us to generate an interactive plan given a SQL query and the underlying database optimizer.

Fulcrum. An interactive plan aims to produce prefix-correct result tuples incrementally. This requires the plan to impose a specific execution order for tuples throughout the execution. Such an execution order may correspond to an explicit query requirement (e.g., `ORDER BY` or `GROUP BY`), or more generally to any attribute that induces a monotonic traversal of the result space. We refer to such an attribute as a *fulcrum*. Importantly, a fulcrum does not assume that data is globally sorted, nor that results are materialized in advance. Instead, the fulcrum is defined with respect to the execution plan: it is an attribute whose access pattern induces the execution order required for incremental result production.

Definition 3.1 (Fulcrum). The *fulcrum* of an interactive plan is an attribute chosen such that query execution begins with an access path (e.g., an index scan) that produces tuples in a sorted order over that attribute. This induced order governs the interactive execution, allowing result tuples to be emitted incrementally in prefix order with respect to the fulcrum.

In Listing 1, we choose *o_orderdate* as the fulcrum. Using an index scan on *o_orderdate* as the first operation of the interactive plan allows us to use index-based grouping for aggregating the sum on *o.o_totalprice* later, making the plan interactive.

Listing 1. `GROUP BY` Example

```
SELECT o_orderdate,
       SUM(o_totalprice)
  FROM orders AS o
GROUP BY o_orderdate;
```

Listing 2. `ORDER BY` Example

```
SELECT o_shippriority,
       o_orderdate
  FROM orders AS o
ORDER BY o_shippriority;
```

In Listing 2, we choose *o_shippriority* as the fulcrum, and we start the interactive plan with an index scan on *o_shippriority*, which imparts an order sorted by *o_shippriority* to the answer tuples outputted. As tuples arrive already sorted on *o_shippriority*, we can answer the query interactively. Even when the query contains no explicit `ORDER BY` or `GROUP BY`, SmartRabbit still

selects a fulcrum and emits results in increasing fulcrum order to maintain the same monotone progress signal for coordination with the blocking plan. Moreover, we can have multiple fulcrum candidates for some queries. For example, if in the first example, we change the query to have a `GROUP BY` on both *o_orderdate* and *o_shippriority*, then either attribute can be chosen as fulcrum. We can also choose a function of an attribute (e.g. `extract(year from` *o_orderdate*`)` if functional indexes are supported by the underlying database system.

To construct an interactive plan for a given fulcrum choice, an access path that induces the required execution order is necessary. In practice, this requires an index on the fulcrum attribute (or an equivalent ordered access method), since otherwise the execution order would have to be imposed via a blocking sort. As mentioned in the introduction, this is an overhead required to support interactive execution more broadly.

Why Left-Deep Tree Plans? A left-deep (join) tree is a binary tree where each join has its left child as either a base relation or another join, and its right child as a base relation. In SmartRabbit, left-deep plans are chosen deliberately to support interactivity rather than to optimize join efficiency. This structure preserves a well-defined execution order across joins, which is essential for incremental result visibility and for avoiding full materialization of intermediate results. When the interactive plan's leftmost scan is an index scan on the fulcrum, this order propagates up the plan. In contrast, bushy join plans allow joins between arbitrary subtrees, increasing flexibility in join ordering and parallelism but sometimes hindering interactive execution. Even if we have a sorted order on the fulcrum in one of the subtrees, it becomes difficult to guarantee that this order is upheld by the other joins in the tree. For example, consider joining (`Orders` $\bowtie$ `Lineitem`) and (`Customer` $\bowtie$ `Nation`) in a bushy shape where we have a sorted fulcrum on Orders. Even if the join of (`Orders` $\bowtie$ `Lineitem`) maintains a sorted order on the fulcrum, we cannot easily guarantee that (`Customer` $\bowtie$ `Nation`) produces tuples in a way that respects this order. Extending SmartRabbit to support bushy subtrees under stronger coordination is left for future work.

3.1 Rules for Creating Interactive Plan

Now that it is motivated that interactive plans are left-deep trees, there are two central challenges in formulating such a plan:
(1) Determining the fulcrum of the plan
(2) Determining the order of operators in the interactive plan

Both challenges are inter-related - the choice of fulcrum dictates how many index scans we will have and strongly influences the join order. In this work, we take a *rule-based* approach to formulating interactive plans, with rules that can be easily integrated into existing optimizer rule frameworks [6, 7]. In a typical optimizer workflow, a query is first translated into a logical plan, which is then rewritten into an optimized logical plan (e.g., via predicate pushdown, projection introduction, subquery decorrelation etc.). The optimized logical plan is subsequently transformed into a physical plan. SmartRabbit intervenes at this stage: it takes the optimized logical plan as input and produces a physical interactive plan. Queries that cannot be decorrelated/flattened by the logical optimizer or queries with ungrouped aggregations are outside the intended scope of SmartRabbit; in such cases, the framework falls back to the optimizer's non-interactive plan. Furthermore, we can apply a set of orthogonal techniques in query unnesting, such as [34], to further broaden our applicability.

Our rules are related to the concept of interesting orders in [40]. The key insight we use is that ordered access paths can be exploited to avoid explicit sorts and enable order-sensitive execution. In SmartRabbit, we first choose the fulcrum that fixes an order. We then treat the chosen order as

a plan-wide invariant since it is required both for early, prefix-correct output and for coordination with the blocking plan during handover.

Table 1 summarizes the rules used to construct interactive plans. The rules are executed in order of their group identifiers, and the process iterates until the plan converges. Rule Group 1 (Fulcrum Selection) identifies the fulcrum. If the query specifies an `ORDER BY`, the first attribute in that clause is chosen as the fulcrum to ensure results are produced in the requested order. Otherwise, a candidate set is formed from base attributes projected in the query. Note that if the query has a grouping clause, then these base attributes will automatically be a part of the grouping keys. Attributes without an index are discarded, since they cannot provide a sorted access path. If there are no such fulcrum attributes, then we simply move to Rule 1.6 for the optimizer to create a non-interactive plan. From the remaining set, we remove attributes without filters, as early filtering reduces work in the interactive plan. Finally, if both primary- and secondary-indexed attributes remain, we prefer primaries, since secondary indexes often require additional primary probes to fetch missing attributes [15].

Rule Group 2 (Plan Shape) enforces a left-deep tree structure. We keep any `GROUP BY` or `ORDER BY` operator at the top of the tree.

Rule Group 3 (Joins) restricts joins to Nested Loop Join (NLJ) or Index Nested Loop Join (INLJ), always preferring INLJ. As we had mentioned in Section 2.2.2, NLJ and INLJ are both interactive join implementations, with INLJ being more efficient as each probing tuple can be resolved immediately with an index lookup, guaranteeing earlier output than a full inner scan. Once the fulcrum is fixed, join ordering is delegated to the optimizer; If we have multiple fulcrum candidates after applying Rule Group 1, interactive plans are enumerated and costed for each, with the least-cost plan implicitly determining the fulcrum.

Rule Group 4 (Aggregation) mandates the use of the SmartRabbit aggregation operator for aggregations. We will describe this operator implementation in Section 4.1 in more detail.

Rule Group 5 (Ordering) similarly prescribes the SmartRabbit ordering operator for queries with `ORDER BY` but no grouping.

Example to show rules in action. Consider the query in Listing 3 that has three tables, four filter conditions and three grouping attributes. We obtain three fulcrum candidates: *l.l_orderkey*, *o.o_orderdate*, and *o.o_shippriority*. Rule 1.1 does not apply as there is no `ORDER BY` clause. Assuming all fulcrum candidates are indexed, Rule 1.3 doesn't eliminate a candidate. Next, following Rule 1.4, we prune *o.o_shippriority*. This leaves us with two candidates: *l.l_orderkey* and *o.o_orderdate*. Rule 1.5 then selects *l.l_orderkey*, preferring its primary index over the secondary index on *o.o_orderdate*.

Listing 3. Example TPC-H Query for Rules Framework

```
  SELECT l.l_orderkey, o.o_orderdate, o.o_shippriority,
         SUM(l.l_extendedprice*(1-l.l_discount)) AS revenue
    FROM lineitem AS l
    JOIN orders AS o ON l.l_orderkey = o.o_orderkey
    JOIN customer AS c ON o.o_custkey = c.c_custkey
   WHERE c.c_mktsegment = 'BUILDING'
     AND o.o_orderdate < DATE '1995-03-15'
     AND l.l_shipdate > DATE '1995-03-15'
     AND l.l_orderkey > 300000
GROUP BY l.l_orderkey, o.o_orderdate, o.o_shippriority;
```

With the fulcrum fixed, Rule Group 2 produces a left-deep plan, with joins following the schema: `Lineitem` to `Orders`, then `Orders` to `Customer`. Rule Group 3 chooses index nested-loop joins for

Table 1. Interactive Plan Rules

Rule Group	No.	Rule
Fulcrum Selection	1.0	If the query contains an ungrouped global aggregation or the optimized logical plan contains unflattened/undecorrelated subqueries, delegate plan construction to the underlying optimizer.
	1.1	If the query has an `ORDER BY` clause, choose the first attribute of the `ORDER BY` clause as the fulcrum.
	1.2	If Rule 1.1 is not applicable, construct a fulcrum candidate set from the base attributes projected in the query.
	1.3	Among candidate attributes, prune those that do not have an index defined on them.
	1.4	Among the remaining candidate attributes, prefer those that have a filter condition.
	1.5	Among the remaining candidates, prefer those with primary indexes.
	1.6	If no viable fulcrum candidate exists, interactive plan is not possible. Delegate plan construction to the underlying optimizer.
Plan Shape	2.1	Always construct the interactive plan as a left-deep tree that begins at the fulcrum index scan, adding all joins as right children along the spine. Place any `GROUP BY` or `ORDER BY` operator at the top of the spine.
Joins	3.1	Restrict joins to Nested Loop Join (NLJ) or Index Nested Loop Join (INLJ).
	3.2	If both NLJ and INLJ are possible, choose INLJ.
	3.3	Once the fulcrum is fixed, delegate join ordering to the database optimizer. If the fulcrum is not fixed, enumerate and cost interactive plans for each candidate and select the least-cost plan, which implicitly determines the fulcrum.
Aggregation	4.1	Use the SmartRabbit aggregation operator to perform both local and global aggregations.
Ordering	5.1	Use the SmartRabbit ordering operator to answer `ORDER BY` queries without grouping.

both, using primary indexes on the join keys. Rule Group 4 then applies the interactive SmartRabbit group-by operator in both to produce aggregate results incrementally.

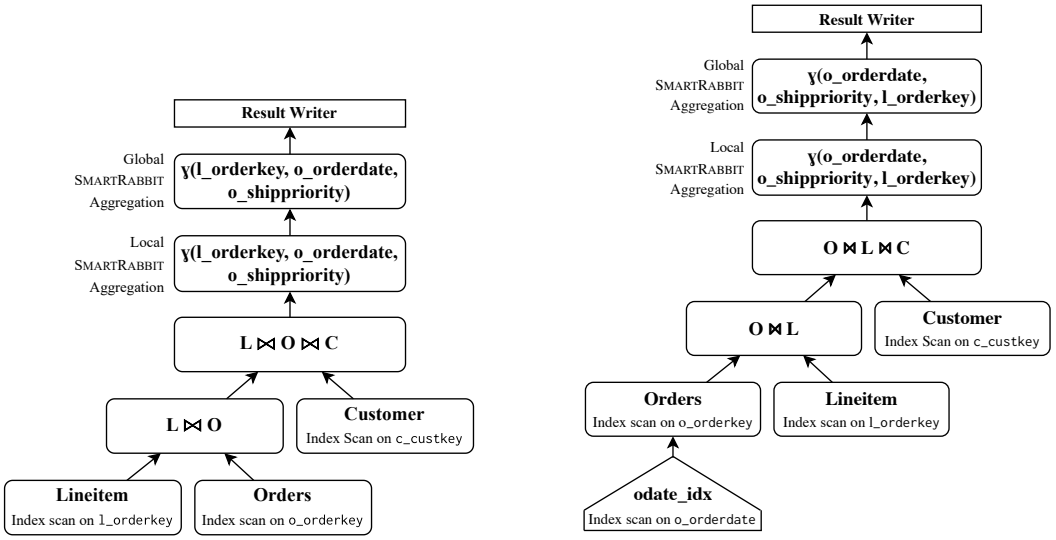
In Figure 2a, we see the plan chosen by SmartRabbit for Listing 3. We also show the alternate plan if we had chosen `o_orderdate` as our fulcrum in Figure 2b. We can see that the alternate plan has an extra index scan, hence explaining our rationale behind Rule 1.5.

4 SmartRabbit Implementation

In this section, we describe the implementation of SmartRabbit in Apache AsterixDB.

AsterixDB's query execution layer is built on Hyracks [8], a dataflow engine for parallel computation. Hyracks supports an operator framework, entitled Algebricks, that implements the necessary operators to support SQL. In AsterixDB, data is distributed over nodes (using hashing on the primary key) and each node contains a predefined number of partitions. As a default, AsterixDB's query processing parallelizes over partitions. AsterixDB's scalability is well established, and it has been adopted as an analytics engine for Couchbase [18].

Supporting Hybrid Query Processing. Figure 3 shows how SmartRabbit is integrated with AsterixDB and how it can be used to answer queries interactively. When a query arrives, two semantically



(a) Chosen interactive plan for Listing 3 (b) Alternate interactive plan for Listing 3

Fig. 2. Interactive plans created by SmartRabbit.

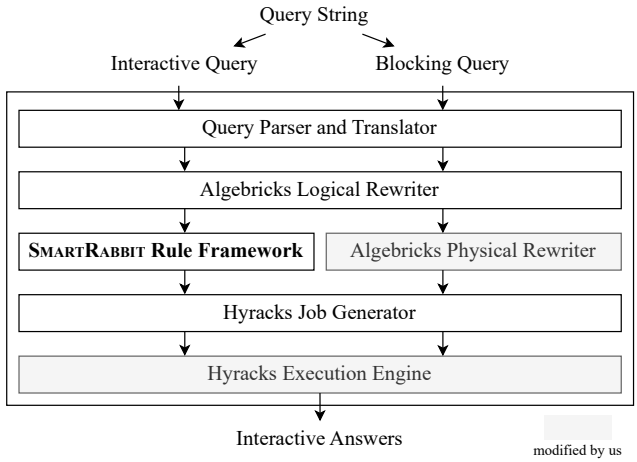


Fig. 3. SmartRabbit Architecture Diagram

equivalent copies of it are created - one interactive and one blocking. Both are parsed and translated into Algebricks [8] logical plans. These plans are further optimized by the Algebricks Logical Rewriter. Then the optimized logical plan for the interactive query is transformed into an interactive physical plan by the SmartRabbit Rule Framework introduced in Section 3. The blocking plan is generated by the Algebricks Physical Rewriter, which has been slightly modified by us to add new operators (Section 4.4) and modify existing operators that would communicate with the interactive execution. Both plans are converted into Hyracks jobs and are submitted to Hyracks for execution. The user receives answers interactively initially from the interactive execution, and then from the blocking execution once the blocking execution is ready to produce all answers. To

support hybrid execution, we modified the Hyracks engine to support simultaneous execution of both interactive and blocking plans. In particular, we added support for localized sorting in different physical operators (Section 4.1), introduced fulcrum informed flushing for interactive execution (Section 4.2) and added mechanisms for the two executions to communicate among each other to maintain the correctness of the results (Sections 4.3 and 4.4). We have also enabled the system to work in a multi-partition parallel execution (Section 4.5). We describe each of these changes below.

4.1 Localized Sorting for Interactive Execution

In the interactive execution, operators consume and process data in fulcrum order. For `GROUP BY`/`ORDER BY` queries, if grouping/ordering is exclusively on the fulcrum attribute, incoming tuples can be processed directly. However, queries often involve multiple grouping/ordering keys. For example, if the grouping keys are A and B, with A as the fulcrum, correctness requires tuples to be ordered by both A and B; since they already arrive ordered by A *by design*, we only need to locally sort tuples sharing the same A value on B. Sorting tuples with the same fulcrum value on a join key or primary key is also useful for reducing random accesses during index nested-loop joins and primary-key lookups. To support these use-cases, we use a lightweight *localized sort* operator that performs just enough local (within-partition) reordering on demand.

Localized sorting as a primitive already exists in database systems; here, we repurpose it to support interactive execution. The key insight is that localized partial sorts can refine the order within each fulcrum group, supporting multi-attribute `GROUP BY`/`ORDER BY` and improving index lookups, without introducing a fully blocking global sort. Thus, when the full required order is not provided by the access path, we construct it incrementally using localized sorting.

Given a fulcrum attribute f and an additional order to achieve, $\vec{o} = (o_1, o_2, \ldots, o_k)$, over one or more non-fulcrum attributes, the operator maintains a buffer B of tuples for the current fulcrum value. When the first tuple arrives, f is stored in *currentKey* and the tuple is placed into B. For each subsequent tuple:

- If its fulcrum value equals *currentKey*, it is appended to B.
- else, sort B by $\vec{o}$, emit all tuples to the next operator, reset B to contain the new tuple, and update *currentKey*.

After all input is consumed, any remaining tuples in B are sorted and emitted. We describe next how the local sorting operator is used to implement aggregation, sorting, and speed up primary index scan operators.

4.1.1 SmartRabbit Aggregation. For our discussion of aggregation, we assume the standard two-level aggregation in distributed settings [41]: each partition performs a local partial aggregation, and these aggregates are merged centrally by a global aggregation.

To make aggregation interactive, we exploit the fact that no further sorting is needed when incoming tuples are already ordered on the grouping keys. We then detect group boundaries and emit aggregates as groups complete. Thus, when the grouping key coincides with the fulcrum, aggregation can proceed directly. When the grouping keys include additional non-fulcrum attributes, we apply a localized sort within each fulcrum group to ensure the required order before aggregation. Localized sorting is needed only for local aggregation: at the global level, local aggregators have already produced tuples ordered by the full grouping key. The local aggregation emits one group at a time as subgroups complete, while the global aggregation merges these partial groups on arrival, avoiding additional sorting or blocking. The algorithm for SmartRabbit aggregation is outlined in the extended version of the paper.

4.1.2 SmartRabbit Ordering Operator. Our approach extends similarly to queries with ordering. When a query includes both `ORDER BY` and `GROUP BY`, we align the grouping keys with the ordering attributes so that aggregation already produces sorted output, leveraging the commutativity of grouping. For queries with only `ORDER BY`, the SmartRabbit Ordering Operator uses localized sorting to incrementally order tuples within each fulcrum group on the remaining ordering attributes and thus deliver the desired order.

4.1.3 Optimizing Primary Index Scan. We found that interactive plans may require multiple index nested-loop joins to join all relations while preserving fulcrum order. These joins often use secondary indexes to probe inner relations, followed by primary-index probes to fetch attributes needed downstream. This extra work can be expensive, as a large primary index may be accessed randomly many times to emit even a single output tuple. To reduce this cost, we use Localized Sort to additionally sort tuples sharing the same fulcrum value on the primary-key attribute. Once tuples are partially clustered on primary-key value, random I/O costs in the interactive plan are reduced substantially.

4.2 Fulcrum-Informed Flushing

In conventional push-based query engines, operators propagate data downstream once their internal buffers (or frames) are filled [32]. For instance, in AsterixDB, tuples are appended to a *frame* until it reaches its capacity, at which point the frame is flushed to the next operator [8]. While this policy ensures high throughput, it also introduces latency for an interactive setting - especially when we want to expedite flushing tuples when a certain fulcrum value is completely seen. SmartRabbit departs from this purely size-driven flushing strategy through a mechanism we term *fulcrum-informed flushing*. Each physical operator in SmartRabbit is made aware of the fulcrum attribute during query planning. As tuples are produced, the operator monitors changes in the fulcrum value; if a new fulcrum value is encountered, the operator proactively flushes any partially filled frame and starts a new one. This allows downstream operators to begin processing results for the completed fulcrum immediately, without waiting for the frame to fill. This approach maintains the push-based execution semantics of AsterixDB while introducing adaptive, content-aware flushing that aligns operator boundaries with the logical structure of the interactive plan.

4.3 Communicating Between Executions

As discussed earlier, in hybrid processing, we need to stop the interactive execution when the blocking execution is ready to produce answers. This is achieved by adding a "signaling logic" to specific operators in the blocking and interactive plans. To maximize interactivity, an interrupt (by signaling logic) is inserted after all the blocking operations of the blocking plan have completed. For example, Figure 4, shows that the blocking execution sends an interrupt to the interactive execution after the rest of the blocking plan is complete and only pipelined (interactive) operations remain.

The interactive execution periodically checks for this signal; once detected, it stops processing safely after emitting its last answer and sends an acknowledgment back to the blocking execution, which then resumes its operation. This handover policy can be sensitive to skew: if the current fulcrum group is skewed, completing the in-progress group may delay termination slightly. While we do not observe a noticeable overhead from this effect in our experiments (Section 5.3), we can also think of a more conservative handover variant for skewed workloads where the interactive execution stops immediately after receiving the signal. Any buffered tuples in the result writer are discarded, and the blocking plan uses the last key value observed so far as the handover filter. In the scenario where the interactive execution completes prior to the blocking execution (which can

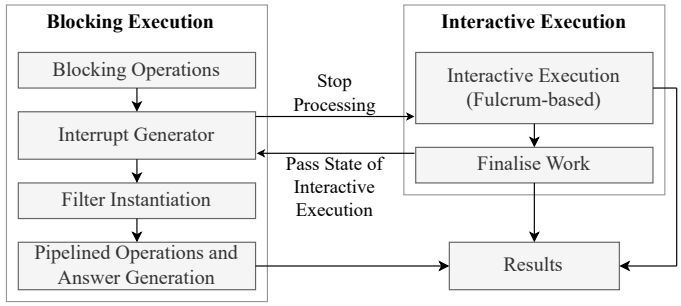


Fig. 4. Interrupt Logic and Handover Filter

happen for some queries), a signal is sent from the interactive execution to the blocking execution to stop the query execution since the interactive execution has already produced all the answers and the blocking execution can stop freeing up resources.

4.4 Handover Filter

Once the interactive execution has stopped, the blocking execution must avoid re-emitting the answers already produced. To do so, a handover filter is added to the blocking plan. During planning phase, we add an empty filter (i.e., a condition that always resolves to true) to the blocking plan at the planning stage and, furthermore, we ensure that the filter is not pushed down the operator tree by the optimizer. When the interactive plan terminates after receiving an interrupt from the blocking plan, it communicates the boundary fulcrum value to the blocking plan: the last fulcrum value (or key tuple for multi-attribute ordering/grouping) for which all result tuples have been emitted, ensuring that no future output can contain a key $\leq$ that boundary. These values, written to a shared buffer between the interactive and blocking plan, enable the blocking plan to dynamically learn the filter value at runtime (Figure 4) which empowers the blocking plan to not emit any answers already emitted by the interactive plan. For example, if the fulcrum is $custkey$ and the last interactive answer has $custkey = 1000$, the blocking execution learns the filter $custkey > 1000$. This guarantees complete coverage while preventing overlap. For queries with composite grouping keys, the learned condition extends to a conjunction over all grouping attributes. For instance, if the grouping keys are $(custkey, orderdate)$ and the last interactive answer has $(custkey, orderdate) = (1000, \text{'1995-03-15'})$, the blocking execution learns the filter $(custkey > 1000)$ OR $(custkey = 1000$ AND $orderdate >$ '1995-03-15'). We emphasize that the sort order on the fulcrum offers a simple and efficient way to avoid duplications compared to the typical duplicate eliminations that are blocking.

4.5 Challenges for Parallel Execution

When executing interactive query plans in a distributed parallel system, we need to solve two challenges: (i) How can we retain sorted orders? And (ii) How can we communicate the handover filter value?

4.5.1 Maintaining a sorted order of inputs. The success of interactive execution relies on preserving the sorted order on the fulcrum attribute. This enables us to answer one fulcrum value at a time and to safely hand over query processing to the blocking execution when needed. When all data resides in a single partition, maintaining this order is straightforward. Thus, downstream operators observe tuples in the same order as produced by the upstream ordered access path. In contrast, in a distributed setting, tuples with the same fulcrum value may reside across multiple nodes. When

these nodes send locally sorted streams to a common receiver, their interleaving can violate the global sorted order, complicating the downstream operators like aggregation.

As an example, suppose we propagate (a, b) tuples from four senders to a single receiver, where each sender locally orders its output by attribute a. Senders 1 and 2 are the fastest and send tuples $(1, 6)$ and $(1, 5)$ first. Before senders 3 and 4 transmit their tuples with $a = 1$, sender 2 sends $(2, 7)$ – thereby breaking the global sorted order at the receiver. This becomes problematic for downstream interactive operators such as `GROUP BY` and `ORDER BY`, which depend on receiving sorted input and would otherwise produce incorrect results.

To address this, we adopt a pull-based merge strategy using merge queues (e.g., heaps). Similar to distributed merge sort, each receiver maintains a sorted priority queue that pulls tuples from sender queues in global sorted order.

4.5.2 Communicating the value of the handover filter. In a parallel system, the result writer operator—the operator responsible for emitting results to the user— is itself parallel, i.e., multiple writer instances run on different partitions. This parallelism makes it impossible to maintain a global sorted order across all writers. If the query does not require ordered output, results could in principle be returned in any order. However, ensuring correctness of the handover filter becomes problematic: when the blocking execution interrupts the interactive plan, the last tuple emitted by a fast partition may not correspond to the true global handover point. A faster partition may have already produced tuples with larger fulcrum values, while a slower partition has yet to emit tuples with smaller fulcrum values.

To guarantee both correctness and consistent termination, we execute the result writer of the interactive plan as a single (unpartitioned) instance. This preserves the global sorted order, provides a well-defined final tuple for the handover filter, and avoids race conditions across partitions. The trade-off is reduced parallelism at the result stage, but this impacts only the final emission operator, not the rest of the interactive pipeline.

5 Evaluation

This section evaluates the performance of SmartRabbit. We describe the evaluation metrics (Section 5.1) and experimental setup (Section 5.2). We then show results across multiple benchmarks and on `LIMIT` queries and perform ablation studies (Section 5.3). We present a comparison against DuckDB at a large scale factor: SF 3000 (Section 5.4). Finally, we analyze the overhead introduced by the dual-execution strategy on concurrently running queries(Section 5.5).

5.1 Metrics

We use the following three metrics to evaluate SmartRabbit against other approaches. ***Time to First Results*** **(TTFR)** [21] that measures the time when the first results of the query are produced since its start. TTFR has been commonly used in prior work to quantify interactivity. ***Interactivity Fraction*** **(IF)** that measures the fraction of answers SmartRabbit returns in its interactive execution, before the blocking execution takes over. ***End-to-End Latency*** that measures the time when the query terminates and all its results have been computed.

The first two measures – TTFR and IF – together provide a holistic picture of the degree of interactivity by indicating how fast and what fraction of results SmartRabbit delivers to downstream operators/users. The latency measure will be used to compare the potential overhead/increase in time compared to the standalone blocking approach to ascertain the cost of interactivity that SmartRabbit supports. Note that for all our experiments, we have plotted TTFR and end-to-end latency relative to the blocking execution time completion (which corresponds to 1.0 in the graphs). The absolute numbers are present in the appendix.

5.2 Experimental Setup

Datasets. We use three popular OLAP benchmarks for our experiments: TPC-H [42], SSB [36], and ClickBench [9].

Queries. We classify queries in our benchmark into 2 categories.

(1) High-cardinality queries (G_1): that produce a large number of result tuples. In the TPC-H benchmark, queries Q2, Q3, Q8, Q9, Q10, Q11, Q16, Q18, Q20 and Q21 fall into this category with cardinality ranging from 730 to 380K at scale factor 10 [2].

(2) Low Cardinality Queries (G_2): that produce a small number of results. In the TPC-H benchmark queries Q1, Q4, Q5, Q7, Q12 and Q22 which produce between 2-10 results, and Q6, Q14, Q15, Q17, and Q19 that produce a single result.

Of the two classes, SmartRabbit is expected to benefit G_1 queries compared to queries in G_2 (specially for the TTFR metric). In particular, G_2 queries that only produce a single result cannot be answered interactively (and, hence, SmartRabbit will not provide any benefit). We remove such queries from our experimental study. Likewise, we only consider 25 (out of 42) queries in ClickBench, and 10 (out of 13) in SSB since the remaining queries cannot be answered interactively (as they produce a single answer). For ClickBench and SSB we do not classify the remaining queries (other than those that produce single answer) into low/high cardinality queries since the smallest query results in 10 or more answers at the scale we study in our evaluation.

For all experiments, we remove `ORDER BY` clauses on derived or computed attributes (e.g., *revenue* in TPC-H), as such orderings cannot be answered interactively using SmartRabbit's current execution strategy. For queries that group or order by a derived temporal attribute (e.g., *o_year*), we instead use the base attribute (*o_orderdate*) to enable index-based access in the interactive plan. In Sections 5.3.1 and the ablation studies, we additionally remove `LIMIT` clauses to focus on scenarios with large result sets. We separately evaluate SmartRabbit's behavior on `LIMIT` queries and compare it against blocking execution in Section 5.3.2.

System Configuration. All experiments were conducted using Apache AsterixDB v0.9.10 (with 150 GB allocated to the JVM) deployed on a 5-node shared-nothing cluster, consisting of one Cluster Controller (CC) and four Node Controllers (NCs) [8]. Each node is a Dell PowerEdge R640 server equipped with dual Intel Xeon Gold 6240R processors (48 physical cores per node, 2.40 GHz) and 192 GiB of DRAM. Each node provides approximately 1.8 TB of local NVMe storage, which was used for AsterixDB data, indexes, and temporary files. The memory allocated to the JVM of each node was set to the default value to half of the available RAM (96 GiB). The buffer cache used by AsterixDB was a default of a fourth of the allocated JVM memory (24 GiB).

Parallelism of System. We configured SmartRabbit to use 16 partitions i.e 4 partitions per each compute node.

Number of Runs. We report the average over 10 runs with one upfront warm-up run to exclude cold-start effects.

Indexes created. The list of indexes that we created in each benchmark is listed in Table 2. As discussed in Section 3, the additional indexes are justified by their role in enabling interactive execution.

5.3 Experimental Results

5.3.1 Performance across benchmarks. ***TPC-H.*** Figure 5 reports relative latency and TTFR for TPC-H at scale factor 100, comparing SmartRabbit against the blocking plans produced by the AsterixDB optimizer. For blocking execution, TTFR equals end-to-end latency since results are produced only upon completion.

[2]when we remove LIMIT and replace *o_year* by *o_orderdate*

TPC-H + SSB		ClickBench	
Index Created	**Queries**	**Index Created**	**Queries**
LINEITEM.L_RETURNFLAG	Q1	ADVENGINEID	Q7
SUPPLIER.S_ACCTBAL	Q2	REGIONID	Q8, Q9
ORDERS.O_ORDERPRIORITY	Q4	MOBILEPHONEMODEL	Q10, Q11
ORDERS.O_ORDERDATE	Q8	SEARCHPHRASE	Q12–14, Q21, Q22, Q24–26
LINEITEM.L_SHIPMODE	Q12	USERID	Q16, Q17
PART.P_BRAND	Q16	CLIENTIP	Q30–32, Q35
ORDERS.O_TOTALPRICE	Q18	URL	Q33, Q34, Q36, Q38
SUPPLIER.S_NAME	Q20, Q21	TRAFFICSOURCEENGINEID	Q39
DATE.D_YEAR(for SSB)	All SSB queries	URLHASH	Q40
		WINDOWCLIENTHEIGHT	Q41

Table 2. Queries and non-PK/FK fulcrum indexes across TPC-H, SSB, and ClickBench.

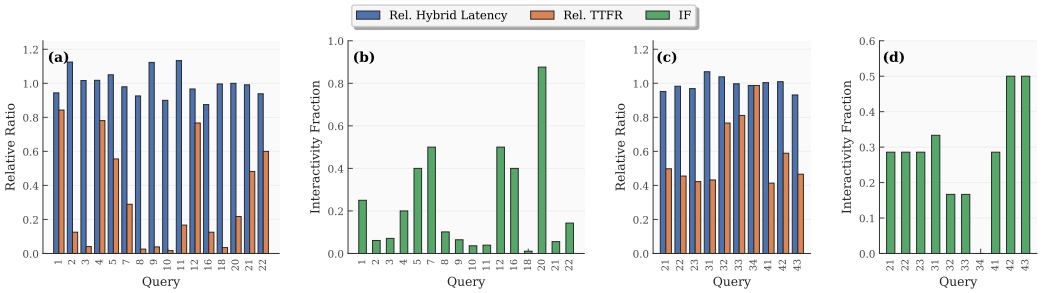


Fig. 5. TPC-H and SSB Performance in SF 100: (a) TPC-H Relative Metrics, (b) TPC-H Interactivity, (c) SSB Relative Metrics, (d) SSB Interactivity

SmartRabbit substantially reduces TTFR across all query groups. For G_1 queries, five queries achieve TTFR below 0.05, and the average TTFR across all 10 queries is 0.12. Even for G_2 queries, we observe an average TTFR of 0.45. At the same time, hybrid execution incurs little completion-time overhead: across all queries, total latency remains within 0.9–1.1× of the blocking plan, a trend consistent throughout this section.

Overall, SmartRabbit retrieves close to a quarter of the total answers on average across both G_1 and G_2 queries (average interactivity fraction (IF) of 0.23), even before the blocking plan retrieves a single answer. The IF for G_2 queries is higher than for G_1, since even a single answer represents a significant fraction of the total result due to the lower cardinality of G_2 queries.

SSB. The relative latency and time-to-first-result (TTFR) for SSB at scale factor 100 are shown in Figure 5(c), with the corresponding interactivity fractions (IF) in Figure 5(d). Across all queries, SmartRabbit achieves even better interactivity compared to the TPC-H benchmark, achieving an average IF of 0.28. All SSB queries include an `ORDER BY` on *d_year*, which naturally serves as the fulcrum for interactive execution. The dataset in SSB spans only seven distinct years and results are produced grouped one year at a time. Thus, the resulting TTFR for SSB is higher compared to that of TPC-H, especially when considering G1 queries, that produce answers continuously. Nevertheless, SmartRabbit consistently produces early results while maintaining an overall latency comparable to blocking execution.

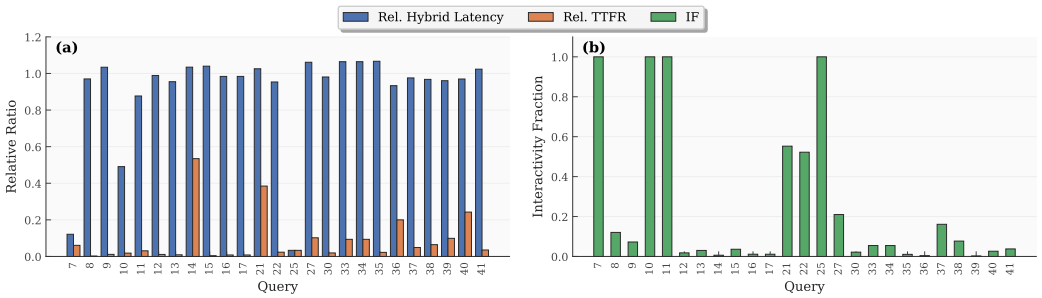


Fig. 6. ClickBench Performance: (a) Relative Metrics, (b) Interactivity (SF 100)

ClickBench. Figure 6(a) reports relative latency and time-to-first-result (TTFR) for ClickBench, while Figure 6(b) shows the corresponding interactivity fraction (IF). For four queries (7, 9, 10, and 25), the interactive plan of SmartRabbit is able to retrieve all the answers for the query even before the blocking plan stops (i.e., achieves **IF = 1**) by selecting an index-only execution plan. In contrast, the blocking optimizer never selects an index-only plan for these queries due to the presence of <> predicates. As a result, hybrid execution not only improves interactivity but also reduces end-to-end latency, yielding speedups of up to **60×** in some cases.

Across all 25 ClickBench queries, SmartRabbit achieves an average IF of 0.24 and an average TTFR of 0.08, indicating that, on average, result production begins within the first tenth of the blocking execution time. These results highlight that SmartRabbit can simultaneously improve interactivity and completion time, particularly when alternative access paths are available but overlooked by conventional optimizers.

5.3.2 Performance on `LIMIT` *queries.* We next explore the use of SmartRabbit for `LIMIT` queries, which are often used as the de-facto mechanism to support interactivity during query processing. We evaluate five representative TPC-H queries (Q3, Q9, Q10, Q11, and Q18) at scale factor 300, measuring the fraction of the blocking execution time required to return the first 10,000 result tuples, as shown in Figure 7. We note that the time taken by AsterixDB for these queries is essentially the same as it would take without `LIMIT`. This is because the `GROUP BY` operator prevents the pushdown of `LIMIT`. In contrast, SmartRabbit returns the first 10,000 result tuples within 4% of the blocking execution time for four of the five queries; only Q11 exhibits a slower result arrival rate. This demonstrates the rapid partial result delivery capability of SmartRabbit. More broadly, the results highlight an additional benefit of SmartRabbit: beyond enabling interactive result inspection,it provides a practical and efficient mechanism for supporting `LIMIT` queries in modern analytical database systems.

5.3.3 Ablation Studies. In this subsection, we study SmartRabbit's behavior as we vary the scale factor and the number of partitions.

Varying the scale factor. In this experiment we vary the scale factor of TPC-H from 10 to 300 and run the queries for 5 scale factors (10, 30, 50, 100 and 300). We plot the average IF and TTFR values for each scale factors in Figure 7(b). We see positive trends for both metrics. The relative TTFR decreases in value, going as low as 0.07 for SF 300 for G_1 queries and 0.37 for G_2 queries. The average IF increases in value, reaching almost 0.28 in SF 300. This shows that **SmartRabbit improves in performance as we keep on increasing the size of the database.**

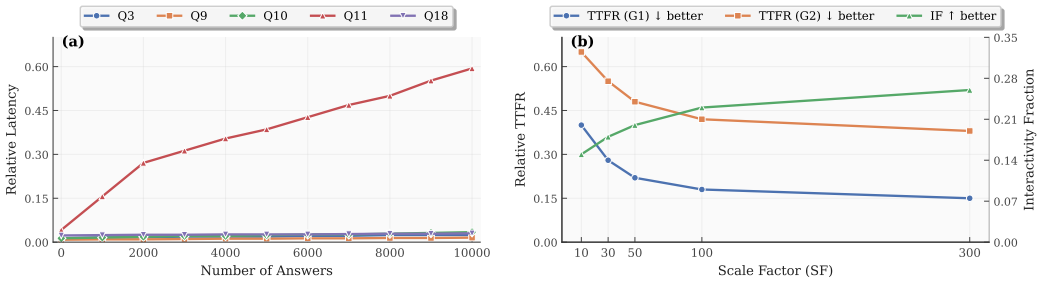


Fig. 7. Performance Analysis: (a) LIMIT Query Performance vs Result Size, (b) SmartRabbit trends across scale factors

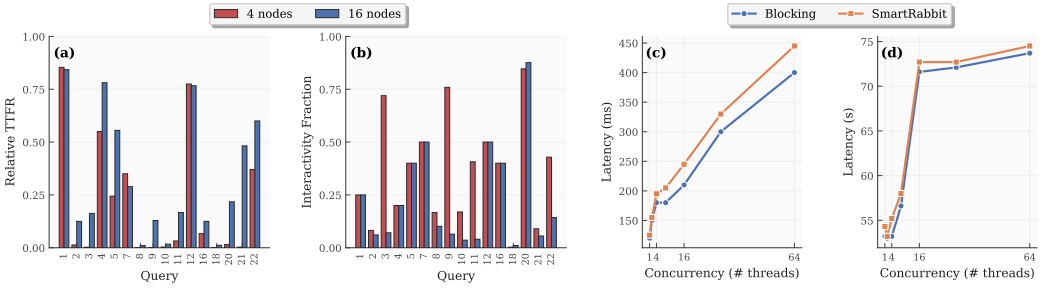


Fig. 8. Scalability and Concurrency of SmartRabbit: (a) Relative TTFR: 4 vs 16 Nodes, (b) IF: 4 vs 16 Nodes, (c) Small Query Latency (Hybrid vs Blocking) , (d) Large Query (Q9) Latency (Hybrid vs Blocking)

Varying the number of partitions. We next study SmartRabbit under different degrees of parallelism to quantify the buffering/coordination costs induced by prefix-correct execution at scale. Figure 8(a,b) shows a clear trade-off when increasing the number of partitions from 4 to 16 at SF 100: the average Relative TTFR shifts from 0.205 to 0.330 (+0.125) and the average IF shifts from 0.370 to 0.232 (-0.138). These shifts reflect the additional buffering and synchronization that arise under higher parallelism. In particular, increased contention in merge queues and the unpartitioned result writer slightly reduce how far the interactive execution can progress before handover, even as the blocking execution benefits from greater parallel throughput. Overall, SmartRabbit maintains robust performance across both settings, with predictable behavior as parallelism increases.

Effect of skew. SmartRabbit's inter-plan coordination can be susceptible to skew. The time between the blocking execution issuing an interrupt and receiving acknowledgment from the interactive execution measures the overhead of waiting for the current group to finish. Across TPC-H SF 10, 100, and 300, the median handover wait time remains in a similar range (1.0,s, 1.5,s, and 1.1,s, respectively), with standard deviations of 3.5,s, 2.9,s, and 3.6,s. This indicates that while most handovers are fast, occasional skewed groups can delay takeover. Even so, at SF 100 the median wait time is only about 1% of the median query runtime, and its relative impact decreases further as dataset size grows.

5.4 Single Node Databases

Modern single-node analytical databases like DuckDB [31] and Umbra [33] have demonstrated remarkable performance, often outperforming distributed systems by large margins when data fits

Query	DuckDB	SmartRabbit	DuckDB/SmartRabbit
Q2	9.87	12.01	0.82
Q3	164.52	**8.12**	20.26
Q8	299.24	**8.74**	34.24
Q9	583.40	**60.08**	9.71
Q10	243.40	**5.42**	44.90
Q11	12.83	**7.24**	1.77
Q16	17.71	35.95	0.49
Q18	263.07	**49.11**	5.36
Q20	222.99	**84.87**	2.63
Q21	303.24	**253.95**	1.19

Table 3. TTFR (seconds) comparison. Bold indicates lower TTFR than DuckDB.

comfortably on a single machine. For smaller scale factors (≤ SF 300 in TPC-H), these databases comfortably return results within seconds, speeds that can be considered interactive. This raises an issue of whether the speed at which such systems can produce answers (despite their blocking query plans) reduces the perceived importance of SmartRabbit. This view, however, overlooks two key considerations. First, SmartRabbit is an engine-agnostic execution strategy; it is not tied to AsterixDB and could, in principle, be integrated into any database system, including single-node engines. Second, as data volume increases, even optimized single-node databases eventually fail to deliver results at interactive rates, e.g., taking several minutes to return the first answers. In contrast, SmartRabbit (despite being built on the lower-throughput AsterixDB engine) could return initial results almost *immediately* (in 10 seconds).

To identify the threshold where single-node performance degrades, we installed DuckDB on a single machine with the same hardware configuration as our AsterixDB nodes, placing the database files on the local NVMe disk. We incrementally increased the TPC-H scale factor until average Time to First Results (TTFR) for DuckDB[3] exceeded several minutes. In our setup, this threshold was SF 3000, where we observed a median TTFR of 174 seconds, with the maximum TTFR being 583 seconds.

We attempted to test DuckDB SF 5000, but the dataset exceeded the capacity of the NVMe disk on our server. When data does not fit on local storage, the corresponding TTFR increases to unacceptable levels - even with SF 3000 with data residing on RDMA-backed NFS and spill files mounted on local NVMe, the median TTFR increases to 561 seconds, with a maximum of 1344 seconds, well above an acceptable 10 minute threshold for interactivity.

To ensure a rigorous comparison, we benchmark SmartRabbit against the best-case scenario for DuckDB: running entirely on local NVMe storage. Table 3 presents a comparison of SmartRabbit's Time to First Results (TTFR) on TPC-H SF 3000 against DuckDB's execution time for queries in the G1 class. DuckDB takes an average time of 212 seconds with a maximum of 583 seconds. In contrast, SmartRabbit takes an average of 52.5 seconds for all such queries for TTFR achieving a maximum improvement of about 45x over when compared to DuckDB.

These results show that SmartRabbit maintains low TTFR at scales precisely where single-node systems become non-interactive, despite being implemented atop a system with a much higher baseline overhead. Since SmartRabbit is an architectural strategy not a system-specific feature, implementing this dual-execution methodology directly within a high-performance engine like DuckDB would likely yield even more significant performance gains.

[3]Due to DuckDB's blocking execution, the TTFR is very close to the query latency.

5.5 Concurrency Study

Since SmartRabbit executes two concurrent plans per query, increasing overall resource consumption, a key concern is whether this degrades the performance of concurrent short, latency-sensitive queries. To evaluate this, we combine a relatively long-running analytical query (TPC-H Q9 at SF 30) with concurrent short-running point, range, and aggregate queries, varying the number of client threads from 1 to 64, doubling them each time. We cap the number of client threads at 64 to reflect the available parallelism on our cluster controller, which provides 48 physical cores, while leaving headroom for query execution and system-level tasks. The long-running Q9 query is executed repeatedly for 10 minutes to ensure steady-state behavior and sustained resource contention, while short queries are issued continuously and selected at random for the same duration.

This setup allows us to quantify the impact of SmartRabbit's additional resource consumption on query latency and throughput relative to conventional blocking execution. Hybrid execution incurs only marginal additional latency for short queries compared to blocking execution (Figure 8(c)), while Q9 latency increases similarly under both execution modes (Figure 8(d)).

6 Related Work

Motivationally, SmartRabbit is related to prior work on improving query interactivity using `LIMIT` clauses and approximate query processing (AQP), as discussed in the introduction. In the context of the former, prior work has proposed extensions to `LIMIT`-based processing, such as rewriting queries into sequences with varying offsets [20] or introducing operators like *rankagg* to reduce tuple processing [23]. These approaches, however, rely on repeated execution, apply to a narrow class of queries, and offer limited benefit in the presence of blocking operators. Approximate Query Processing (AQP) [1, 2, 35] improves interactivity by trading accuracy for latency, producing fast approximate answers that are refined over time. While effective for dashboards and visualization, AQP does not provide guarantees of correctness and does not support incremental production of exact results for general multi-stage analytical workflows. In contrast, SmartRabbit targets interactive evaluation of *exact* query results for a broad class of SQL analytics, without requiring approximation or repeated execution.

From a technical and methodological perspective, SmartRabbit is loosely related to prior work on adaptive query processing and optimization, such as tuple-routing architectures like Eddies [5, 38], which dynamically route tuples to different operators based on the expected execution strategy. A more recent example is Plaque [24], which learns predicates from downstream operators (e.g., aggregations and joins) at runtime and pushes them down dynamically to reduce computation. SmartRabbit shares this adaptive flavor, making flexible decisions during query execution based on runtime state, though its goals and mechanisms differ substantially. SmartRabbit can also be improved with better join orders. POLAR addresses this by pipelining multiple join orders and selecting the best order at runtime [22]. While our current implementation is limited to left-deep trees, integrating POLAR-style adaptivity could enable SmartRabbit to explore alternative join orders.

Progressive query optimization (POP) [25] is another work closely related to SmartRabbit. It introduces a `CHECK` operator to detect significant runtime cardinality deviations and trigger re-optimization when beneficial, with the key ability to reuse work already computed before the switch; this idea has also been adopted by DataBricks [45]. In contrast, SmartRabbit does not re-optimize plans for cost: it coordinates an interactive and a blocking execution based on operator readiness to improve interactivity while preserving correctness.

We also note that running multiple plans concurrently has prior precedent, e.g., the competition model in DEC Rdb/VMS [4], which executes alternative plans in parallel and selects a single winner

based on observed progress. SmartRabbit differs in that both plans are intended to contribute to the final answer.

An orthogonal but complementary line of work is *incremental query processing* [44], where queries are executed over extended periods with planned pauses. The goal is to process available data early, often during off-peak times, and complete the remaining work near the deadline to improve cost and resource efficiency. Differential dataflow [26] supports incremental maintenance of query results under data updates, while we focus on large-scale static data.

MotherDuck's Instant SQL provides query previews by executing queries as the user types, leveraging local execution, AST-based rewriting, and caching [30]. Instant SQL focuses on front-end responsiveness during query formulation, using caching and data sampling to achieve fast responses, similar to previous AQP approaches.

7 Conclusion and Future Work

For large datasets, blocking query plans may delay output for minutes before producing any results. We presented SmartRabbit, a system for interactive query processing that emits early results while a latency-optimized blocking execution proceeds in the background. This enables downstream consumers to receive useful output even for traditionally blocking queries such as grouped aggregations. Built on AsterixDB, SmartRabbit shows that hybrid execution can provide early results while preserving nearly the same end-to-end latency as conventional blocking plans. Experimentally, SmartRabbit outperforms a `LIMIT`-based approach on blocking execution by several degrees of improvement, improves further at larger scale factors, and maintains interactivity as the number of partitions increases. Importantly, it remains interactive even at scales where highly optimized single-node database systems fail to return results quickly.

We see several directions for future work. We are exploring extending SmartRabbit's dual-execution strategy to other database engines. While our system currently processes the full dataset in both the interactive and blocking plans, cost-aware partitioning could improve efficiency by splitting the data between the two. Another direction is to coordinate the interactive and blocking executions, for example, via scheduling to reduce contention or by pushing down filters learned in one execution to the other, exploiting their different access orders. Making the optimizer aware of interactive execution and developing cost models is exciting too.

Lastly, this work studies how interactivity can be achieved rather than how it can be utilized. A promising direction is user studies to better understand and prioritize interactivity needs.

8 Acknowledgements

We gratefully acknowledge the support of the following grants NSF 2540011, 2420846, 2419844, 2245372, 2113930, 2133391, 2008993, 1952247. We also acknowledge support from the Hasso Plattner Research Center at UCI. The authors thank Michael Carey and Ian Mason for their generous support and invaluable assistance with the AsterixDB codebase and system configuration, as well as for their guidance in designing and executing the SmartRabbit experiments. Their insights into the system were instrumental in enabling a careful and meaningful evaluation. We also thank the anonymous reviewers for their thoughtful and constructive feedback, which helped significantly improve the clarity and positioning of the paper's contributions. Finally, we acknowledge the help of Sahil Makhijani with several experiments related to DuckDB.

References

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 487–498. ACM, 2000.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, page 29–42. ACM, 2013.

[3] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.

[4] G. Antoshenkov. Dynamic query optimization in rdb/vms. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 538–547. IEEE, 1993.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 261–272. ACM, 2000.

[6] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 221–230. ACM, 2018.

[7] V. Borkar, Y. Bu, E. P. Carman, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 422–433. ACM, 2015.

[8] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1151–1162, 2011.

[9] ClickHouse, Inc. ClickBench: A benchmark for analytical databases. https://benchmark.clickhouse.com, 2026. Accessed: 2026-03-15.

[10] Databricks. Agent bricks. https://www.databricks.com/product/artificial-intelligence/agent-bricks, 2026. Accessed 2026-03-15.

[11] J. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 299–310. 2002.

[12] 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 *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 313–324. OpenProceedings.org, 2019.

[13] D. Fisher, S. Drucker, and M. Czerwinski. Trust Me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 1673–1682. ACM, 2012.

[14] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 102–111. ACM, 1990.

[15] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[16] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.

[17] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 171–182. ACM, 1997.

[18] M. A. Hubail, A. Alsuliman, M. Blow, M. J. Carey, D. Lychagin, I. Maxon, and T. Westmann. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.*, 12(12):2275–2286, 2019.

[19] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[20] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):1–58, 2008.

[21] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 299–310. ACM Press, 1999.

[22] D. Justen, D. Ritter, C. Fraser, A. Lamb, N. Tran, A. Lee, T. Bodner, M. Y. Haddad, S. Zeuch, V. Markl, and M. Boehm. POLAR: adaptive and non-invasive join order selection via plans of least resistance. *Proc. VLDB Endow.*, 17(6):1350–1363, 2024.

[23] C. Li, K. C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 61–72. ACM, 2006.

[24] Y. Lin and S. Mehrotra. PLAQUE: automated predicate learning at query time. *Proc. ACM Manag. Data*, 2(1):46:1–46:25, 2024.

[25] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 659–670. ACM, 2004.

[26] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Biennial Conference on Innovative Data Systems Research (CIDR), Online Proceedings*. www.cidrdb.org, 2013.

[27] Microsoft. Databases in fabric documentation. https://learn.microsoft.com/en-us/fabric/database/, 2026. Microsoft Learn. Accessed: 2026-03-15.
[28] Microsoft. What is Copilot in Fabric? https://learn.microsoft.com/en-us/fabric/fundamentals/copilot-fabric-overview, 2026. Accessed 2026-03-15.
[29] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 251–262, USA, 2004. IEEE.
[30] MotherDuck. Instant SQL is here: Speedrun ad-hoc queries as you type. https://motherduck.com/blog/introducing-instant-sql/, April 2025. Accessed: 2026-03-15.
[31] H. Mühleisen and M. Raasveldt. DuckDB. https://duckdb.org, 2026. Version 1.4.4.
[32] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.
[33] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *Conference on Innovative Data Systems Research (CIDR), Online Proceedings*. www.cidrdb.org, 2020.
[34] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *Business, Technologie und Web (BTW)*, volume P-241 of *LNI*, pages 383–402. GI, 2015.
[35] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. C. Reed. Interactive analysis of web-scale data. In *Biennial Conference on Innovative Data Systems Research (CIDR), Online Proceedings*. www.cidrdb.org, 2009.
[36] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference (TPCTC), Revised Selected Papers*, pages 237–252. Springer, 2009.
[37] M. Raasveldt and H. Mühleisen. DuckDB: an embeddable analytical database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1981–1984. ACM, 2019.
[38] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 353–364. IEEE Computer Society, 2003.
[39] Reuters. Snowflake partners with OpenAI in $200 million AI deal. https://www.reuters.com/business/snowflake-partners-with-openai-200-million-ai-deal-2026-02-02, 2026. Accessed: 2026-03-15.
[40] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 23–34. ACM, 1979.
[41] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 104–114. ACM, 1995.
[42] Transaction Processing Performance Council (TPC). TPC-H: A decision support benchmark. https://www.tpc.org/tpch/, 2026. Accessed: 2026-03-15.
[43] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
[44] Z. Wang, K. Zeng, B. Huang, W. Chen, X. Cui, B. Wang, J. Liu, L. Fan, D. Qu, Z. Hou, T. Guan, C. Li, and J. Zhou. Tempura: a general cost-based optimizer framework for incremental data processing (journal version). *VLDB Journal*, 32(6):1315–1342, 2023.
[45] M. Xue, Y. Bu, A. Somani, W. Fan, Z. Liu, S. Chen, H. V. Hovell, B. Samwel, M. Mokhtar, R. Korlapati, A. Lam, Y. Ma, V. Ercegovac, J. Li, A. Behm, Y. Li, X. Li, S. Krishnamurthy, A. Shukla, M. Petropoulos, S. Paranjpye, R. Xin, and M. Zaharia. Adaptive and robust query execution for lakehouses at scale. *Proc. VLDB Endow.*, 17(12):3947–3959, 2024.

Received October 2025; revised January 2026; accepted February 2026