

AXR: Reusing Intermediate Results for LLM Agents in PostgreSQL

Runguo Li

University of Illinois Urbana-Champaign

Urbana, IL, USA

runguo.ai@gmail.com

ABSTRACT

LLM agents answer analytical questions by exploring databases through SQL, and they are commonly run several times per question for reliability. On large data, the database then takes a large share of the agents' time, and much of its work is redundant: attempts repeat their siblings' statements, often while these are still running, and recompute the expensive parts of their own earlier statements under new statement text. We present AXR, a PostgreSQL extension that materializes the results of plan subtrees as queries run and substitutes them into later plans. Its resolved signatures bind the run-time parameters of PostgreSQL's parameterized nested loops and sub-query plans to the expressions they carry, so the subtrees that dominate agent queries can be matched. A statement can wait for a result that a concurrent attempt is still building, without risk of deadlock. A result is kept only if recomputing it costs more than reading it back, and reuse stays correct under concurrent writes through the same server. We replay twelve workloads recorded from one agent with two LLMs. On TPC-H the agents largely recall the benchmark queries, so we also recorded them on the same data with renamed tables and rewritten questions. AXR reduces database time by 6.5–51.0% at scale factor 10 and by 23.9% on a TPC-H workload at scale factor 100. Where attempts converge, a statement cache that shares running statements obtains most of this saving (21.6% at scale factor 100). Reusing subtrees adds 21.3–23.0 points on every recording of one held-out set of TPC-DS questions, with either LLM, but only 2.0–8.9 on a second. In 776,114 checked statement executions and 1,958 random queries compared value by value, AXR returned no wrong result.

1 INTRODUCTION

LLM agents are becoming users of databases in their own right. Given an analytical question, an agent explores the schema and the data through SQL, inspects intermediate results, and revises its queries until it settles on an answer. Because a single attempt is unreliable, LLM systems commonly make several attempts at the same question and select among their answers [24, 32], and agents can run these attempts concurrently. This style of access has been called *agentic speculation*, and it has been argued that data systems should be redesigned for it [21].

When the data is large, the database becomes a bottleneck for such agents. In our recordings of agents answering TPC-DS and TPC-H questions at scale factor 10 in PostgreSQL, a median attempt spends 18–56% of its time waiting for the database rather than for the LLM (3–27% with a slower LLM). Much of that work

is redundant. Different attempts at the same question converge on the same expensive joins and aggregations, and an attempt that revises its query recomputes the expensive parts of its own earlier statements. An offline analysis of the TPC-H traces shows that reusing plan subtrees could save 58–75% of the database time of the median question, while reusing identical statements could save only 26–42%. These traces, however, overstate convergence: the agents largely reproduce the public TPC-H queries from memory.

Existing reuse techniques capture only part of this redundancy. Statement-level result caches and workflow-level deduplication [30] see only identical statements or calls. That suffices where attempts repeat each other's statements verbatim, as they largely do on TPC-H, but agent statements often differ even when their expensive parts do not. Recycling of intermediate results [7, 15, 22] reuses subtrees, but agent workloads defeat a direct application of it in three ways. First, PostgreSQL plans rarely contain equal subtrees for equal computations: columns are referenced by position, join keys and correlated values are passed through run-time parameters, and sub-queries are planned separately. Binding these parameters adds 10.5–26.4 points to the saving of database time in our ablation; reusing a result under a different projection brings no measurable gain (it changes the saving by –4.1 points on N4 and by nothing measurable elsewhere). Second, concurrent attempts ask for results that a sibling is still computing; without in-flight sharing, the saving of subtree reuse on TPC-H N4 drops from 56.9% when the traces are replayed one statement at a time to 28.8% when the attempts run concurrently as recorded. Third, intermediate results of agent queries reach gigabytes, and keeping the wrong ones wastes more than it saves.

We present AXR, a PostgreSQL extension that reuses intermediate results across the statements and attempts of LLM agents. AXR signs plan subtrees with *resolved* signatures, which describe every column by the base-table column it derives from. This lets it bind the run-time parameters of nested-loop joins and sub-queries to the expressions they carry, which makes the subtrees that dominate agent queries reusable. It lets a statement wait for a result that another session is still building, without any risk of deadlock. It admits a result only if rebuilding it costs more than reading it back, and it remains correct under concurrent writes through the same server. Admission addresses only part of the third challenge: it keeps results that are cheap to recompute out of the cache, but on TPC-DS the results that pass still exceed the storage budget, and choosing which of them to keep remains open. AXR needs no change to PostgreSQL and consists of about 3,200 lines of C.

We evaluate AXR by replaying twelve recorded agent workloads with their original timing and concurrency, and we first check whether the workloads are what they seem. On TPC-H they are

not: 59–60% of the agents’ final queries are near copies of the specification’s queries. We therefore also recorded agents on the same data behind renamed tables and columns, with rewritten questions; there, no final query is a near copy. We recorded the renamed TPC-H questions and two held-out sets of TPC-DS questions twice more, once with a second LLM, to see how much a result depends on one recording. AXR reduces database time by 6.5–51.0% on these workloads, and by 23.9% at scale factor 100, where the database takes 97.8% of the agents’ time. We separate three layers of reuse: a cache of SQL text, a statement cache that matches plans with AXR’s signatures (C1i), and subtree reuse. Matching plans rather than text matters everywhere: statements whose normalized text had already been issued provide at most 73% of the statement cache’s saving, and only 12–38% on nine of the twelve workloads. Where attempts converge, as on TPC-H, the statement cache gets most of the saving (44.3% and 47.3% against 49.1% and 51.0%). Much of it depends on sharing results that are still being computed: without that, concurrent attempts on N4 lose 44% of the reuse that a sequential replay realizes. Subtree reuse matters on one held-out set of TPC-DS questions, where it adds 21.3–23.0 points on all three recordings, with either LLM; most of that gain comes from a few questions in which agents revised one statement around expensive building blocks without finding the answer. On a second held-out set it adds only 2.0–8.9 points, and on the other workloads at most 5.7. In 776,114 checked statement executions and 1,958 random queries compared value by value, AXR never returned a wrong result.

Our contributions are:

- A characterization of LLM-agent workloads on analytical databases, based on twelve traces of one agent with two LLMs over TPC-H and TPC-DS: a check for benchmark recall, which inflates the redundancy of TPC-H traces; when database time matters; how much of it is redundant and where the redundancy comes from (mostly across attempts on TPC-H, largely within attempts on TPC-DS); and why uniform sampling is not an alternative (§2).
- The design of AXR (§3): resolved signatures that bind PostgreSQL’s run-time parameters, deadlock-free sharing of in-flight results, cost-based admission and consistency under writes.
- An implementation as a PostgreSQL extension, tested against PostgreSQL without AXR on every result and on random queries (§4, §5.8).
- An evaluation on twelve workloads with and without recall, re-recorded with a second LLM, at two scale factors, that separates what matching statements by plan rather than text, sharing running statements and reusing subtrees each contribute, and shows why concurrency loses reuse (§5).

2 AGENTS ON ANALYTICAL DATABASES

To see what agents do to an analytical database, we recorded them answering TPC-H and TPC-DS questions, with several attempts per question.

2.1 Setup and workloads

Agent. We use a minimal tool-calling SQL agent in the ReAct style [33]: at every step the LLM runs SQL statements against the database and sees their (truncated) results, or submits a final query. All exploration happens through SQL, in at most 15 LLM steps per attempt; a step may issue several statements. The LLM is a commercial model served through an API (DeepSeek’s *flash* model, temperature 0.8); for two re-recordings we used DeepSeek’s *pro* model with thinking disabled. Each question is attempted N times ($N = 16$ in workload N16 and 4 in all others, as the names indicate), which is how LLM systems commonly trade compute for reliability [24, 32]. A pool of four workers takes the attempts question by question, so at most four attempts are in flight at once and they are mostly siblings; each workload is recorded and replayed on its own, against its own cache. We record every statement with its timing and the LLM latency of every step.

Databases and questions. TPC-H and TPC-DS at scale factor 10 run in PostgreSQL 16 (16 GB and 23 GB of data and indexes by `pg_database_size`). For TPC-H, the questions are the 22 business questions of the specification with its validation parameters; N16 uses eight of them, drawn at random with a fixed seed (Q1, 3, 7, 11, 12, 19, 20, 21). Because the agents turned out to recall these queries, we also recorded H4o, a renamed version of N4 (§2.2). For TPC-DS, we kept the 90 of 99 official queries that finish within 120 s and split them into three disjoint sets of 30; an LLM rewrote each query as a natural-language question. The first set (DS4) served to design AXR’s policies; the second (DS4b) was collected later and used only for evaluation; the third (DS4c) was recorded last, after all other workloads had been evaluated. To measure how much a result depends on one recording, we recorded H4o, DS4b and DS4c twice more, with the same questions, question order and settings: once with the same LLM (H4oR2, DS4bR2, DS4cR2) and once with the *pro* model (H4oPro, DS4bPro, DS4cPro). Agentic text-to-SQL benchmarks such as Spider 2.0 [19] target enterprise databases, many of them in cloud warehouses such as BigQuery and Snowflake; we used TPC-H and TPC-DS because they can be generated at any scale in PostgreSQL. Table 1 summarizes the twelve recorded workloads. The TPC-DS questions are hard for this agent and our grader compares result sets strictly, so per-attempt accuracy is low (7.5–13.3%); the statements are nonetheless exactly what the agents issued while trying, and their exploration is longer: 17.0–20.9 statements per attempt against 5.8–10.5 on TPC-H.

2.2 Do the agents recall the benchmark?

The TPC-H specification and its queries are public, so an LLM may have memorized them. Attempts that recall the same reference query converge for a reason that does not carry over to new questions. Before analyzing redundancy, we therefore checked for recall with two measures fixed in advance. The first is the share of final queries that contain an identifier that occurs in the question’s reference query but neither in the schema nor in the question, such as a column alias; as a control, we count identifiers that only the other questions’ reference queries contain. The second is the token similarity between a final query and the reference query, with literals replaced by placeholders. Table 2 shows both. We count recall as indicated if the first share exceeds its control by 20 points

Table 1: Recorded agent workloads (scale factor 10, PostgreSQL 16). DB share: median over attempts of database time over database plus LLM time, replayed without AXR (mean of three replays). Acc.: accuracy per attempt. [†] pro model.

Workload	Quest.	Attempts	SQL/att.	DB share	Acc.
TPC-H N4	22	88	5.8	49%	84.1%
TPC-H N16	8	128	6.5	56%	70.3%
Renamed H4o	22	88	7.8	53%	85.2%
H4oR2	22	88	8.2	53%	87.5%
H4oPro [†]	22	88	10.5	27%	88.6%
TPC-DS DS4	30	120	18.3	29%	9.2%
TPC-DS DS4b	30	120	17.7	31%	12.5%
DS4bR2	30	120	17.0	26%	13.3%
DS4bPro [†]	30	120	18.9	11%	10.0%
TPC-DS DS4c	30	120	18.3	21%	12.5%
DS4cR2	30	120	18.5	18%	8.3%
DS4cPro [†]	30	120	20.9	3%	7.5%

Table 2: Benchmark recall. Ref.-only: final queries with an identifier found only in the question’s reference query (control: identifiers of other questions’ reference queries). Sim.: median token similarity of final and reference query; ≥ 0.9 : near copies; Siblings: median similarity of final queries of attempts at a question.

Workload	Ref.-only	Control	Sim.	≥ 0.9	Siblings
TPC-H N4	56.8%	8.0%	0.91	59%	0.97
TPC-H N16	48.4%	14.8%	0.91	60%	0.91
Renamed H4o	5.8%	1.2%	0.48	0%	0.90
H4oR2	6.0%	2.4%	0.46	0%	0.90
H4oPro	4.9%	1.2%	0.51	0%	0.93
TPC-DS DS4	17.9%	26.3%	0.19	0%	0.52
TPC-DS DS4b	20.2%	18.3%	0.34	0%	0.65
DS4bR2	19.6%	29.9%	0.33	1%	0.60
DS4bPro	24.4%	20.7%	0.43	1%	0.70
TPC-DS DS4c	28.7%	40.4%	0.28	0%	0.66
DS4cR2	29.6%	40.8%	0.25	1%	0.66
DS4cPro	28.9%	47.4%	0.36	0%	0.71

or at least 25% of the final queries are near copies (similarity ≥ 0.9). On TPC-H, 56.8% (N4) and 48.4% (N16) of the final queries contain such identifiers, against controls of 8.0% and 14.8%, and 59–60% are near copies of the reference query: the agents largely reproduce the specification. On TPC-DS, whose questions an LLM rephrased from the official queries, no recording indicates recall.

We therefore recorded H4o on the same TPC-H data. The agent sees only views with neutral table and column names (for example `purchase_line` for `lineitem` and `gross_amount` for `l_extendedprice`) in a database named `shop`, through a read-only role that can access nothing else. It answers the 22 questions rewritten from the reference queries, with every filter, output column and ordering stated in the new names. Agent, LLM and settings are those of N4. On H4o and its re-recordings, no final query is a near copy of the renamed reference query, and at most 6.0% contain a reference-only identifier (controls 1.2–2.4%).

Attempts at the same question still write similar queries (median similarity 0.90, as on TPC-H): when a question fully determines the query, attempts converge without recall. We treat N4 and N16 as the regime in which attempts repeat each other, and the TPC-DS and renamed TPC-H recordings as workloads without recall.

2.3 When does database time matter?

Database time matters only when queries are expensive relative to LLM calls. On the BIRD text-to-SQL databases [20], which are small, the same agent spent a median of 0.6% of each attempt in the database. At scale factor 10 in PostgreSQL, it is 49–56% on TPC-H and 18–31% on TPC-DS with the flash model, and lower with the pro model, whose calls take longer (27%, 11% and 3%; Table 1). A faster engine moves the boundary rather than removing it. We re-executed each of the 707 distinct statements of N4 and N16 in isolation on DuckDB [25] with five threads (PostgreSQL used a leader and four workers) and recomputed each attempt’s database share from the per-statement times and the recorded LLM times. The statements that take at least a second in PostgreSQL ran a median of 18.7 $\times$ faster, and the median database share of an N4 attempt fell from 44% (PostgreSQL, computed the same way) to 5.7% at scale factor 10; at scale factor 100 it was back at 46% (N16: 45%). The problem that AXR addresses therefore arises where the data is large relative to the engine’s speed (we discuss columnar engines in §6).

2.4 How much work is redundant, and where does it come from?

We re-executed every distinct statement of the TPC-H workloads in isolation, captured its plan, and computed how much database time each question could save under increasingly general forms of reuse. These are upper bounds that ignore the cost of materializing and reading results; on TPC-H they are inflated by recall. For the median question of N4, identical statements account for 26% of database time, statements that are equal after SQL canonicalization for 34%, and reuse of join and aggregation subtrees for up to 58% (70% if scans are included); for N16 the figures are 42%, 52% and 75% (80%). Reuse of plan subtrees thus offers 1.4–2.2 times what matching whole statements can obtain. In total rather than median database time, the subtree bounds are 60% (68%) on N4 and 65% (72%) on N16. They count only reuse within a question, and they assume that every result is available when it is requested. Under the recorded concurrency, a statement cache that also shares statements that are still running realizes most of what AXR realizes on TPC-H (§5.2).

The source of the reuse differs between the benchmarks. Figure 1 classifies the cache hits of a sequential replay by which statement produced the reused result. On TPC-H, most of the avoided work comes from a *sibling attempt* of the same question (91% on N4, 64% on N16): different attempts converge on the same expensive joins and aggregates. On TPC-DS, the largest share comes from the *same attempt* (75% on DS4, 52% on DS4b): as an attempt revises its query over many steps, successive statements recompute expensive subexpressions of its own earlier statements. On H4o, siblings again dominate (72%, against 19% from the same attempt),

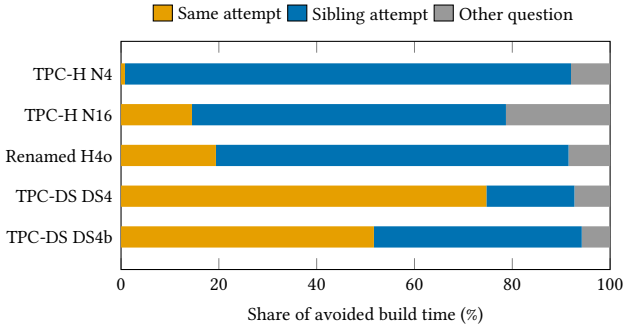


Figure 1: Where reused results come from: share of the build time avoided by reuse, by the statement that produced the result (sequential replay with every result admitted).

so the convergence of TPC-H attempts is not only recall. Results produced for other questions contribute 6–8%, except on N16 (21%). A statement-level cache captures only the part of these patterns in which whole statements repeat, because the statements differ even when their expensive parts do not.

Timing matters as well. Most reuse is separated by LLM think time, but when attempts run concurrently, a sibling often needs a result that another attempt is still computing. In concurrent replays of TPC-H N4, 22% of the reuses were of results that were still being built when they were requested; on the other workloads 2–12% were. On N4, sharing such in-flight results adds 20.2 points to the saving of database time (§5.4).

2.5 Why exact reuse?

Agents tolerate imprecision in some exploratory queries, which suggests answering them approximately. On a 1% uniform sample, the 101 sampleable exploratory aggregate queries of the TPC-H workload ran a median 42× faster, but only 19 stayed within 10% relative error, and the median of the maximum relative error was 0.99, mostly because groups disappeared and top- k results changed; at that bound, sampling would save 6% and 5% of the database time of N4 and N16 (0% and 1.2% for the median question). Stratified or outlier-aware samplers target exactly these failures; we did not test them. Generalized materialization, such as storing an aggregate without its filter for later queries with other filters, would add 9.6 and 8.3 points over exact subtree reuse for the median question of N4 and N16, of which only 0.8 and 3.3 are reachable without materializing results that no query asked for. AXR therefore reuses exact results only.

2.6 Challenges

Turning the redundancy above into saved time inside an unmodified PostgreSQL raises four problems. (C1) *Identifying equivalent subtrees* (§3.2, §3.3). PostgreSQL plans refer to columns by position, pass join keys and correlated values through run-time parameters, and represent sub-queries as separate plans, so equal computations rarely produce equal plan trees. (C2) *Concurrency* (§3.4). Sibling attempts request results that are still being computed. (C3) *What to keep* (§3.5). Intermediate results on TPC-DS reach gigabytes, and

some are cheaper to recompute than to read. (C4) *Correctness* (§3.6). A reused result must equal what PostgreSQL would return, including under concurrent writes.

3 THE DESIGN OF AXR

AXR is a PostgreSQL extension that materializes the results of plan subtrees while queries execute and substitutes them into later plans that contain an equivalent subtree. It needs no change to the PostgreSQL source code: it is loaded through `shared_preload_libraries`, installs a planner hook, a utility hook, an executor-start hook and a transaction callback, and adds two custom scan nodes to the executor.

3.1 Overview

Figure 2 shows AXR at work on two attempts. When a session plans a SELECT, AXR first lets the standard planner produce the final plan and then walks it top-down. For every *candidate* subtree (joins, aggregations, sorts, window aggregates, unions, sub-query scans, base-table scans, and parallel Gather subtrees as a whole) whose estimated cost exceeds a threshold, it computes a signature (§3.2) and consults a hash table in shared memory:

- If a usable result exists, the subtree is replaced by an *AXR Reuse* node that streams the materialized tuples from a file. The original subtree stays as a fallback child that runs only if the file is gone or does not match at execution time; the walk does not descend further.
- Otherwise, if the subtree is admitted (§3.5), it is wrapped by an *AXR Store* node that passes every tuple through to its parent and appends it to a file. Descendants are rewritten first, so one statement can reuse some subtrees and store others.

A Store publishes its entry only when its child has been consumed completely; a rescan, an early stop by a parent (for instance a LIMIT above a join), an error or a transaction abort discards the partial file. A direct child of a LIMIT is never stored: it is rarely consumed completely, and wrapping it would hide the row bound that the LIMIT passes down to a sort. Files hold length-prefixed PostgreSQL minimal tuples in a directory of the database cluster; they are read sequentially and are visible to every session. AXR writes no catalog entries, so it also serves read-only transactions.

Each entry records its state (BUILDING, READY, REJECTED or EVICTED), the backend that builds it, row and byte counts, the measured build time, the planner’s estimates, and the task that produced it. Agents label each connection with its question and attempt through the setting `axr.task`; the label is used only for diagnostics and for the workload analysis of Section 2, never for matching, so a result built for one question can serve another.

3.2 Resolved signatures

A natural signature for a plan subtree is a canonical serialization of the subtree. Our first prototype used exactly that, with query-specific fields removed. It failed in a way that is specific to how PostgreSQL represents plans: column references above a scan are positional. A join refers to its inputs’ columns as “attribute k of the outer child”, and the child’s output list is trimmed to the columns that the rest of *this* query needs. Two agent queries that join the

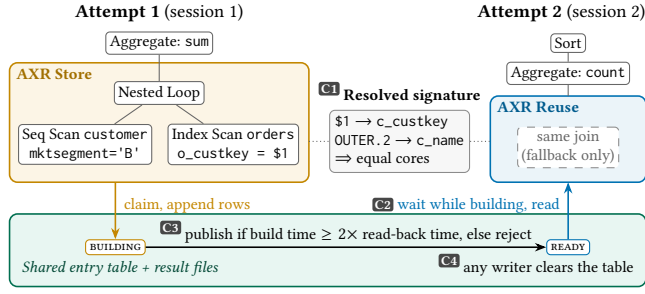


Figure 2: AXR on two concurrent attempts (illustrative plans); C1-C4 are the challenges of §2.6. A Reuse runs its fallback subtree only if the build is discarded.

same tables with the same predicates but select different columns therefore produce different serializations for every node above the first point where their projections differ. In our first prototype, replacing them with the resolved signatures and column-subset reuse described next raised the realized saving of a sequential TPC-H replay from 23.8% to 34.1% of database time (measured on that prototype; later builds do not contain it).

Resolution. AXR resolves every column reference down to a base-relation column, identified by the position of its scan node in a canonical traversal of the subtree and its attribute number; the scan itself contributes its relation OID, its filter and, for index scans, its index and index conditions, and every signature includes the OID of the database, since relation OIDs are unique only within one. Expressions are resolved recursively, so a predicate on a join output becomes a predicate on base columns. The *core* of a signature covers the node types, relations, resolved predicates, join and grouping keys, sort orders and the shape of the tree, but not the subtree’s output list. Each output column is described separately by the hash of its resolved expression. An entry’s key is the core hash together with its ordered column hashes. Fields that carry only parse locations or planner bookkeeping (such as PostgreSQL 16’s nulling-relation markers, whose information the core already contains through the join types) are blanked before hashing.

Why equal signatures mean equal results. The core and the column hashes are computed from everything that determines a subtree’s result: the database and the relations it reads, the operators, every predicate, join and grouping key and expression, each written in terms of base columns, and the values bound to its parameters. The blanked fields are those that PostgreSQL derives from the others or uses only for error messages. Two subtrees with equal cores therefore compute the same function of the same relations, and §3.6 ensures that their snapshots see the same committed data. Three exceptions are excluded rather than signed: non-immutable functions (below), the input of a LIMIT, which is not consumed completely, and subtrees that reference parameters of prepared statements, whose values may change between executions of a generic plan, or contain operators that AXR cannot describe (for example foreign, function and sample scans); such subtrees are

unsignable. A stored result may present rows in a different but valid order where the plan’s order is not total.

Lookups compare hashes (the core’s in 128 bits, two 64-bit hashes of PostgreSQL’s hash_bytes_extended with different seeds; each column’s in 64 bits plus its type, type modifier and collation), which are not cryptographic. AXR therefore does not take equal hashes for equal descriptions: each result file begins with the full descriptions of its core, its columns and the sub-plans nested in the core, and a reader compares them with its own before it reads a row. On any difference, a Reuse computes its subtree and a Store passes its input through, so a collision costs a recomputation, not a wrong result.

Column-subset reuse. A reusing subtree matches an entry if the cores are equal and the entry’s columns include every column it outputs. The Reuse node then projects with a column map. Several variants of one core (different column sets) may coexist.

Look-alike columns. Projection-insensitive signatures raise a correctness issue that our tests exposed (§4): an aggregate such as count(*) resolves to the same expression in every aggregation node, so a statement that cross-joins several count(*) sub-queries (TPC-DS Q88) produced output columns with equal hashes, and the column map of a Reuse could serve the first count for all of them. AXR therefore stamps every aggregate, window function and GROUPING expression with the position of the node that computes it, and that stamp is part of both the core and the column hashes.

What is never cached. Statements that are not plain SELECTs, row-locking reads, scrollable cursors and statements that contain any non-immutable function are left untouched: a stable function such as current_date may return another value in a later statement.

3.3 Binding run-time parameters

A subtree can be signed only if its result depends on nothing outside it. In plans for agent queries, the most common obstacle is not an exotic operator but PostgreSQL’s own parameter passing: when we compared the reuse that a development build realized on a TPC-H replay with an offline upper bound, the subtrees that could not be signed mostly contained one of the two constructs below.

Parameterized nested loops. For an index nested-loop join, PostgreSQL passes the current outer row’s join key to the inner index scan through an executor parameter. The inner side alone is not self-contained, but the whole join is: AXR binds each such parameter to the resolved outer expression that the join assigns to it (the innermost binding wins when joins are nested), so the inner side’s index condition becomes a condition on base columns. Memoize nodes, which cache inner results, are transparent for the signature.

Sub-queries. A correlated sub-query appears as a SubPlan expression whose plan reads parameters set from the outer row. AXR resolves the sub-plan with those parameters bound to the resolved arguments and describes the expression by the resulting signature; parameters that carry the comparison value of IN/ANY tests become placeholders. An uncorrelated sub-query (an InitPlan) is computed once and referenced through a parameter, which AXR

replaces by the signature of the init plan. A scan of a common table expression is described by the signature of the CTE’s plan. The plans inside init plans and CTEs are themselves rewritten, so their results can be stored and reused; correlated sub-plans, which run once per outer row, are not.

Other operators. Sub-query scans, Append/MergeAppend (union all and partitioned tables, with per-branch outputs), Result, window aggregates (partitioning, ordering and frame), grouping sets with GROUPING(), Group and bitmap index combinations are supported. Plans below a Gather run in parallel workers, so AXR treats a Gather subtree as a unit and never rewrites inside it. For every candidate that is not reused or stored, AXR records the reason (for example, which construct made it unsignable) and exposes it per statement; this diagnostic is how we found the gaps above.

In sequential TPC-H replays with the evaluation build (§4; on the shared server of §5.5), subtree reuse without either construct, everything admitted, saves 34.4% on N4 and 32.1% on N16, less than a statement cache whose plan matching supports both (42.4% and 48.4%); with both and cost-based admission it saves 58.3% and 61.2% (admission alone adds at most 6.9 points in the concurrent ablation of §5.5).

3.4 Sharing results that are still being computed

When several attempts of a task run at the same time, a sibling often asks for a subtree while another attempt is still computing it, and a cache that only knows finished results computes the subtree twice. On N4 (§5.4), this lowers the saving of subtree reuse from 56.9% in a sequential replay to 28.8%, and that of a statement cache from 45.5% to 23.7%.

AXR makes an entry visible from the moment a backend *claims* it, which happens when its Store node is first executed. A planner that finds a matching BUILDING entry plans a Reuse, as for a finished one. At executor start the Reuse waits on a condition variable in shared memory until the entry becomes READY, then reads its file; if the build is discarded instead, the Reuse runs its fallback subtree. A Store that finds its key claimed by another backend when it starts executing (two plans raced) waits likewise, then serves the file as a reader, or builds the result if the other builder gave up.

No deadlocks. Waiting is safe only if no cycle of backends can wait for each other. AXR lets a backend wait only while it owns no unfinished build; a backend that has claimed an entry never waits. Every waits-for edge among AXR’s own waits therefore points from a backend without builds to a builder, and builders have no outgoing edges of this kind. A cycle through PostgreSQL’s locks, which its deadlock detector cannot see through a condition-variable wait, would need a builder to wait for a lock held by a waiter. Statements that use AXR are read-only and hold only the shared relation locks of reads, which conflict only with the exclusive locks of writers; and a writer, when it starts, clears the cache and wakes every waiter, which then falls back to its own subtree (§3.6). A timeout bounds a wait in case a builder stalls; on timeout the waiter falls back as well. We set it to 300 s while developing on N4 at scale factor 10 and did not tune it; at scale

factor 100 it is too short (§5.6). A waiter reads only published, complete files.

3.5 Admission and storage

Materializing has a price: the Store writes every tuple, and a later reuse pays for reading the file. Results that are large but cheap to recompute gain little or nothing. On TPC-H, the two largest entries of an admit-all run (full scans of lineitem, 8.2 GB, and of orders, 1.9 GB) were about as fast to recompute with a parallel scan as to read back. AXR admits in two stages:

- (1) *At planning*, a subtree is materialized only if its estimated cost per estimated output byte is at least a threshold (`axr.min_density`). We set it to 10^{-3} cost units per byte, about twice the break-even point derived from measured time per planner cost unit and read-back throughput.
- (2) *At publish*, the measured build time must be at least twice the time to read the result back (estimated at 1 ms per MB; a Reuse node reads 0.71–0.75 ms per MB on our servers, §5.7). Otherwise the file is dropped and the key is remembered as REJECTED, so it is not built again.

The build time is measured inside the stored subtree only. In a pipelined plan the parent’s work is interleaved with the child’s, and counting it would overstate what a reuse saves. Stored results are bounded by a byte budget; when a new result does not fit, the least recently used finished entries are evicted. A size-capped, frequency-aware alternative that we derived from traces did not generalize to held-out questions (§6).

3.6 Consistency

Reusing a result across statements and sessions is correct only if the data it was computed from did not change between the snapshot of the statement that built it and the snapshot of the statement that reads it. Agent exploration is read-only, but AXR must remain correct when writes happen. It enforces the following rules.

- A transaction becomes a *writer* when it first executes a statement that modifies data (any non-SELECT, a data-modifying CTE, or a utility command other than pure reads and session settings). Writers are detected when they execute, not when they are planned, so cached plans and prepared statements are covered.
- The cache is cleared when a writer starts and again when it ends. While any writer is open, no statement stores or reuses, in the writer’s own session or elsewhere.
- Statements under REPEATABLE READ or SERIALIZABLE isolation do not use AXR: their transaction snapshot may predate the newest results, and under SERIALIZABLE a Reuse would also bypass the predicate locks that serializable snapshot isolation takes on the relations a statement reads.
- A statement uses AXR only if its snapshot sees every recently ended writer. AXR keeps the transaction ids of the last 64 committed writers in a ring and checks them against the statement’s snapshot. This closes the window in which a snapshot taken just before a writer committed is used after the writer has ended. A writer’s id enters the ring in AXR’s end-of-transaction callback, which PostgreSQL runs after the commit has become visible; a snapshot *S* that does

not see a committed writer W was therefore taken before W entered the ring, and S fails the check as long as W is in it. W leaves the ring only after 64 later entries; for S to pass then, all 64 must be visible to S , i.e. 64 writers must have become visible before S was taken yet entered the ring after W . The check can thus miss a writer only if 64 writers are, at the same moment, between making their commit visible and running the callback. While they are, they still count as open writers, so no statement stores or reuses.

- A build that spans a clear finds its entry gone when it tries to publish and is discarded.

Together these rules ensure that every finished entry was built after the end of the last writer, from a snapshot that saw that writer, with no writer open during the build, and that every reader uses a snapshot with the same property. Builder and reader therefore observe the same committed data. The guarantee covers writes executed through the instance on which AXR runs. Changes that arrive by replication (WAL replay on a physical standby or the apply worker of a logical subscription) are not detected, so AXR must not be used on replicas or on subscribed tables. Clearing everything on a write is coarse; invalidation per relation is a natural refinement that read-mostly agent workloads did not require.

4 IMPLEMENTATION AND TESTING

AXR is about 3,200 lines of C for PostgreSQL 16.6: plan resolution and signatures (1,070 lines), and planning, execution, shared state and policies (2,110 lines). It uses only public extension interfaces: the planner, utility and executor-start hooks, a transaction callback, custom scan nodes, a shared-memory hash table protected by a lightweight lock, and a condition variable for in-flight waits. Every feature has a run-time setting, so a single binary serves all configurations of the evaluation: the reuse granularity (`axr.mode`: off, statement, subtree), plan-shape coverage (`axr.coverage`), column-subset reuse (`axr.superset`), admission (`axr.policy`), in-flight sharing (`axr.inflight`) and the byte budget. In statement mode AXR considers only the root of each plan; it then acts as an exact result cache that shares all of AXR’s signature machinery, which makes it a strong baseline. The performance runs of Section 5 use the *evaluation build*, which compares only hashes on a hit, except DS4c and the closed-loop replay; the *final build* adds the comparison of descriptions of §3.2, whose cost §5.7 measures.

Testing. Correctness checking is part of every experiment rather than a separate phase. We use a PostgreSQL build with assertions enabled and five kinds of tests.

- (1) *Differential replay.* Recorded agent statements are executed in order with AXR enabled, and each result is compared, as a multiset, with that of the same statement without AXR.
- (2) *Benchmark queries.* Each official TPC-H and TPC-DS query runs once without AXR, once storing and once reusing; both AXR results must equal the first.
- (3) *Look-alike constructs.* Twelve targeted queries with output columns that look alike but differ (argument-less aggregates in different sub-queries, window functions, grouping sets, set operations) or with stable functions.

- (4) *Concurrency.* Simultaneous identical queries (one build, the others wait and read), a session that plans against an in-flight build, cancellation of a builder while others wait, and six sessions running 144 overlapping statements.
- (5) *Writes.* Seven scenarios of Section 3.6: a committed update from another session, an open writing transaction and its rollback, a prepared update executed from a cached plan, a data-modifying CTE, a build that is running while another session commits, `REPEATABLE READ`, and `DDL`.

In addition, every statement of every evaluation run is hashed, and the hashes are compared with those of the runs without AXR (§5.1). On the final build we repeated all five kinds of tests and the random-query test of §5.8, and the first four on TPC-H with every description comparison forced to fail (350 readers recomputed): no result differed.

The tests found two bugs that the evaluation workloads alone would not have revealed. The assertion-enabled build caught a tuple-slot type mismatch that a release build would have turned into wrong data. The official-query test found the look-alike column bug of §3.2 on TPC-DS Q88, after 68,790 checked statement executions on the affected builds had not triggered it. Both are fixed; all reported numbers come from builds with the fixes, except where Section 5 states otherwise.

5 EVALUATION

We replay the recorded agent workloads of Section 2 against PostgreSQL in several configurations.

5.1 Methodology

Replay. Each recorded attempt is replayed on its own connection, starting at its original offset from the beginning of the workload. Between statements the connection sleeps for the recorded LLM latency of the step (its *think time*), so attempts overlap as they did when they were recorded, and the statements run for real. No LLM is called during replay, which makes runs repeatable and lets every configuration see exactly the same statements. Replay is open loop: the recording ran attempts on four workers, question after question, but an attempt that ends sooner in replay does not let the next one start sooner; §6 compares a closed-loop replay. We report the total database time of all statements, the mean end-to-end time of an attempt (database time plus think time), and the mean over questions of the end-to-end time of the longest attempt, the question’s *makespan* if its attempts start together. With four workers they cannot all start together (N16 has sixteen per question), so this is a proxy that leaves out queueing.

Configurations. One AXR binary, the evaluation build (§4), provides all configurations (the final build for DS4c and the closed-loop replay), so that each step of the ladder adds one element of the design:

- C0** PostgreSQL without AXR.
- C1** Statement cache: whole statements only, full plan coverage, everything admitted, no in-flight sharing.
- C1i** C1 with in-flight sharing: a statement waits for an identical statement that is still running.

- C2 Exact-column subtree reuse: resolved signatures but only for identical column sets, without parameterized plans or sub-queries, everything admitted.
- C3 C2 with column-subset reuse (§3.2).
- C4 C3 with full plan coverage (§3.3).
- C5 C4 with cost-based admission (§3.5).
- C6 C5 with in-flight sharing (§3.4): the full AXR.

C1 is a strong form of statement-level result caching, and C1i adds to it the coalescing of concurrent identical statements; C1i is the baseline against which we measure what subtree reuse adds. C2 is a recycler-style step of AXR’s own ladder, not a reimplement of a published recycler [15, 22]: it does not match column-granular intermediates or subsume selections. All configurations use a 24 GB result budget, LRU eviction and a minimum subtree cost of 10^5 planner units. These settings and the admission thresholds (§3.5) were fixed during development on the N4 trace, before any other workload existed, and were not tuned afterwards.

Protocol. Every configuration runs three times. Within each repeat block the order of configurations is random, the cache is emptied before every run, and each workload starts with a discarded warm-up run without AXR. We report the mean of the three runs and, for reductions, the minimum and maximum over the three *paired* comparisons with the C0 run of the same block. The three runs replay the same recording, so they capture measurement noise within one experiment; variation between recordings is probed by two re-recordings per question family (§2.1). We wrote down hypotheses and decision rules before each experiment and report every outcome; the raw results of all experiments, including earlier runs on a shared server, are in our anonymized artifact. Our main criterion credits subtree reuse with a gain on a workload only if AXR (C6) saves at least 5 points of C0 database time more than C1i on average over the three repeat blocks, and more in every block. We call such a gain *replicated* if it also holds on both re-recordings of the same questions; this tests robustness across recordings and LLMs, not across question sets.

Correctness. Every statement result of every run is hashed as a multiset of rows, with numbers rounded to two decimals, dates in ISO form and strings trimmed; row order is ignored. A result of a run with AXR must equal one of the results that the same statement produced in the runs without it. Statements with LIMIT may legitimately differ; we compare how often they do with how often runs without AXR differ from each other (§5.8).

Platform. We use three identical servers, each with two 32-core Intel Xeon Platinum 8358 processors (128 hardware threads), 2 TB of memory and a local NVMe RAID, running Ubuntu 24.04 and PostgreSQL 16.6. Each server runs one PostgreSQL instance per benchmark with 32 GB of shared buffers, 256 MB of work_mem, up to four parallel workers per Gather node and eight in total, and no JIT compilation; AXR stores its results in the database directory on the RAID. All configurations of a workload run on the same server: N4, N16, H40, DS4 and DS4b on the first, the re-recordings (together with H40 and DS4b once more, and the ablation of §5.5) on the second, and the scale-factor-100 experiment on the third; DS4c ran on the first, DS4cR2 and DS4cPro on the second. The servers are shared with other users. Before every run we record

the load average; a workload whose runs started at a load average of 40 or more was run again in full once the load had stayed below 32 for 30 minutes, and we report the repeated runs; §5.2 reports the first runs (load up to 96). The ablation, DS4c and the closed-loop replay instead waited before every block until the load was below 32 at three checks 10 minutes apart and were not re-run. The largest load average before a reported run was 29, except for 3 runs of DS4c and the closed-loop replay (at most 58).

Availability. Our anonymized artifact contains AXR’s source code; the recording agent with its prompt and the question sets; the twelve recorded workloads (every statement with its timing, the LLM latency of every step, and the attempts’ final queries); the replay, correctness and analysis scripts; and the raw results of every run reported here. The agents used the API model identifiers deepseek-flash and, for the second LLM, deepseek-v4-pro with its reasoning mode (“thinking”) disabled; we did not record finer model versions. The artifact is at https://osf.io/nfpx6/?view_only=0a1e13b58521432f8625330e56513c2f.

5.2 Overall results

Table 3 shows the twelve workloads. AXR reduces database time by 6.5–51.0%, and the reduction is stable across repeat blocks. Because database time is only part of an attempt’s time (Table 1), attempts finish 2.4–35.7% sooner on average.

Which mechanism provides the saving depends on the workload. On TPC-H, where the attempts largely repeat each other’s statements, a statement cache with in-flight sharing (C1i) obtains most of it: 44.3% of AXR’s 49.1% on N4 and 47.3% of 51.0% on N16. In-flight sharing is what makes the statement cache effective on N4, where it adds 20.7 points to C1: the four attempts of a question run concurrently and often need a result that a sibling is still computing (§5.4). On renamed TPC-H, too, whole statements account for nearly all the saving on two of the three recordings ($\Delta = -0.9$ and -0.1 points); on the second LLM’s recording, subtree reuse adds 5.7 points (paired 5.7–5.8), meeting our criterion there only.

On the held-out question set DS4b, subtree reuse provides most of the saving. AXR saves 29.5% of DS4b’s database time against 8.2% for C1i, 21.3 points more (paired 20.8–22.1). The gain replicates on both re-recordings, 23.0 points with the same LLM and 22.7 with the second, and on the second server (21.4). Its size depends on contention: in the high-load first runs (§5.1), Δ was 39.3 on DS4b, 10.4 on DS4 and within 1.8 points of Table 3 elsewhere; we report the low-load values. On the design set DS4, subtree reuse adds 2.9 points, below our criterion.

On the third question set, the gain does not carry over. Subtree reuse adds 2.0 points on DS4c and 5.0 on DS4cR2 (4.99 before rounding), both below our criterion, and 8.9 on DS4cPro, which meets it on that recording only (not replicated). DS4c ran on the final build and under the per-block load gate (§5.1); one of its runs (C1i, third block) started at a load average of 58 and was not re-run. That block gives DS4c’s largest paired difference (5.8 points); the other two average 0.1. On the seven workloads that miss the criterion, AXR and C1i differ by -0.9 to 5.0 points. Of the five that meet it, two lie outside DS4b, and both were recorded with the pro model, which also issues more statements per attempt (Table 1).

Table 3: Reduction of database time vs. C0 (% , mean of three runs; for C6, paired minimum and maximum in brackets). Δ : C6 – C1i in points of C0 database time, with its paired range. Ete, Mk: reduction of the mean attempt end-to-end time and question makespan with C6. GB: data written per C6 run. Txt: share of C1i’s saving from statements whose normalized text had started earlier in the run (§5.2). Differences are computed before rounding. [†] second LLM. Re-recordings ran on the second server (there, H4o and DS4b gave $\Delta = -0.8$ and 21.4), without C5. [‡] final build (§4); DS4c on the first server.

	Workload	C0 DB (s)	C1	C1i	Txt	C5	C6 AXR	Δ (C6 – C1i)	Ete	Mk	GB		
TPC-H	N4	881	23.7	44.3	49	28.8	49.1	[48.2, 49.9]	4.7	[4.2, 5.3]	33.2	37.3	3.1
	N16	1664	41.5	47.3	73	45.2	51.0	[50.7, 51.2]	3.7	[3.3, 4.1]	35.7	12.5	5.5
Renamed TPC-H	H4o	2540	11.7	15.2	15	13.8	14.4	[13.8, 15.1]	-0.9	[-1.1, -0.4]	11.6	15.4	46.5
	H4oR2	3909	13.4	14.6	30	-	14.5	[14.1, 14.9]	-0.1	[-0.4, 0.2]	12.6	11.9	11.5
	H4oPro [†]	1831	15.7	21.2	58	-	26.9	[26.6, 27.2]	5.7	[5.7, 5.8]	14.7	9.6	18.9
TPC-DS	DS4	1976	9.3	10.2	12	13.8	13.1	[12.0, 14.7]	2.9	[2.5, 3.5]	7.1	1.7	47.1
	DS4b	2474	5.7	8.2	28	28.8	29.5	[28.7, 30.9]	21.3	[20.8, 22.1]	18.5	16.7	46.7
	DS4bR2	3281	12.6	12.2	14	-	35.2	[34.4, 36.1]	23.0	[22.2, 24.2]	23.6	22.1	55.4
	DS4bPro [†]	1534	12.6	13.1	17	-	35.8	[35.1, 36.4]	22.7	[21.7, 23.3]	12.3	15.7	37.9
TPC-DS [*]	DS4c	3725	3.0	4.5	21	-	6.5	[4.0, 9.1]	2.0	[0.0, 5.8]	4.5	4.3	30.1
	DS4cR2	3517	2.8	3.5	38	-	8.5	[7.7, 9.8]	5.0	[2.2, 7.2]	3.9	1.8	39.6
	DS4cPro [†]	1126	4.4	4.6	37	-	13.5	[12.0, 15.1]	8.9	[7.9, 10.0]	2.4	2.1	16.7

The value of subtree reuse may thus depend on how an agent revises its queries as well as on the questions; we have not tested this. On DS4cPro, though, the database takes only 3% of an attempt’s time, and attempts finish 2.4% sooner. Subtree reuse is not free: where it adds little, AXR still writes 46.5–47.1 GB per run (H4o, DS4), while C1i writes at most 0.5 GB. In database time it costs at most 0.9 points where it does not help, so enabling it trades storage for a large gain on question sets like DS4b; disabling it adaptively is future work.

Could an agent framework obtain C1i’s saving by caching SQL text outside the database? Column Txt of Table 3 classifies each statement C1i served by whether one with the same normalized text had started earlier in the run. Such textual repeats provide 49–73% of C1i’s saving on N4, N16 and H4oPro, but only 12–38% on the other nine workloads. There, most statements that C1i serves differ in text from the statement that produced the result, for example the same join written with a common table expression or with other aliases; AXR’s plan-based matching recognizes them, a textual cache would not.

5.3 When does subtree reuse pay off?

We broke the difference between AXR and C1i down by question after seeing these results. On every recording of DS4b, a few questions account for most of it: 4, 4 and 2 of 30 questions make up 80% of Δ on DS4b, DS4bR2 and DS4bPro. They are largely the same questions. On DS4b, questions 23 and 47 contribute 9.3 and 6.3 points, and question 23 is also among the two largest contributors on both re-recordings, with either LLM.

In these questions, an attempt first computes the building blocks of the answer in separate statements (for question 23, the items that sell frequently and the totals of the best customers), then combines them in one statement with common table expressions and revises that statement several times. Every revision recomputes the blocks, and revisions rarely repeat each other’s text, so mostly subtree reuse serves them: on DS4b, C1i saves 11% of question 23’s database time and AXR 76%. None of the 40 attempts at the questions that make up 80% of Δ ended correct, and attempts that ended

incorrect yield 96–100% of Δ , about their share of the database time (93–96%): subtree reuse served attempts that did not find the answer, which the database cannot tell apart from the others while they run.

On DS4 and on renamed TPC-H, in contrast, a single question dominates the database time: question 51 takes 44% of DS4’s, and question 9 takes 49% of H4o’s (55% of H4oR2’s). No configuration saves much on them (AXR: 9% and -1%), and on H4o subtree reuse adds little even when the trace is replayed one statement at a time (38.4% with C5 settings against 37.0% for the statement cache). In question 51, for instance, the four attempts compute the expensive running totals in different ways, and each attempt usually changes that computation from one statement to the next. In these recordings, subtree reuse thus paid off where an agent revised its statements around expensive parts that stayed the same. On renamed TPC-H, the outcome for the same questions changed with the LLM (Table 3). This is a hypothesis drawn from a few questions, not a tested rule. A rival explanation is the origin of the questions: the official queries of questions 23 and 47 are built from named common table expressions, so the questions may invite this decomposition, though that of question 51 is too and its attempts did not settle on stable blocks. On DS4c, recorded after this analysis, question 39, whose official query is built the same way, is among the two largest contributors on every recording (0.9, 1.0 and 7.0 points).

5.4 Concurrency and in-flight sharing

Replayed one statement at a time, AXR with cost-based admission but without in-flight sharing (C5) saves 56.9% of N4’s database time; replayed with the recorded concurrency, the same configuration saves 28.8%. To find out why, we logged every store, reuse and eviction in a sequential replay and in one concurrent run per configuration of N4 and H4o (the other experiments use three), and classified each reuse of the sequential replay by what happened to the same statement in the concurrent run, weighting it by the time it saved sequentially. On N4, the concurrent C5 run realized 56% of that saving; 44% was lost because the result was still being built by

a sibling attempt when the statement needed it. With in-flight sharing (C6), 93% was realized, and the remainder was lost because the concurrent timing reversed the order of producer and consumer. On H4o, C5 realized 59% and lost 40% in flight, and C6 realized 96%.

In-flight sharing thus recovered nearly all of the reuse that concurrency deferred. The remaining gap on H4o between the sequential saving (38.4%) and the concurrent one (13.5% in this run) is not lost reuse: the concurrent C6 run saved 416 s through reuse, close to the sequential 440 s. But without AXR the same statements take 2.2 times as long concurrently as sequentially (1.3 times on N4), so the same absolute saving is a smaller fraction of a larger total.

5.5 Where the savings come from

Table 4 adds the elements of the design one at a time, each on top of those above it, on N4, N16 and the three DS4b recordings. All runs used the second server and the load gate of §5.1 (load average at most 25.7 before a run); C6 gained 4.4 and 4.5 points over C1i on N4 and N16 and 21.3–23.6 on the DS4b recordings, as in Table 3. An earlier ablation of N4 and N16 on a shared server, also used in §3.3 and §6, found smaller steps (in the artifact). We call a step *measurable* if its three paired values have the same sign and its mean is at least one point.

- *Coverage* (C3→C4) adds 10.5 and 13.5 points on N4 and N16 and 19.2–26.4 on the DS4b recordings: without support for parameterized nested loops and sub-queries, the subtrees that matter most cannot be signed, and subtree reuse without it (C2, C3) saves less than the statement cache, which has it. It alone is measurable on every DS4b recording.
- *Column-subset reuse* (C2→C3) changes the saving by −4.1 points on N4 and by nothing measurable on N16 or the DS4b recordings. On N4, the loss comes from a statement that column-subset reuse made start before a sibling had built the result C2 reused, which C3 recomputed. The gain that resolved signatures with column subsets brought our first prototype (§3.2) thus likely comes mostly from resolution.
- *Cost-based admission* (C4→C5) cuts the data written and adds 6.9 points on N4, not measurably on N16 (0.7). On DS4b, DS4bR2 and DS4bPro it adds 5.8, 1.5 and 0.8 points, measurable only on DS4b; its effect depends on the recording.
- *In-flight sharing* (C5→C6) adds 19.9 and 6.2 points on N4 and N16 (20.2 and 5.7 in Table 3, with −0.7 and 0.6 on DS4 and H4o) and, depending on the recording, 1.2, 1.1 and 1.7 on DS4b, DS4bR2 and DS4bPro, measurable on DS4b and DS4bPro.

5.6 More data and more parallelism

Scale factor 100. We replayed N4, recorded at scale factor 10, with its think times against TPC-H at scale factor 100, with a 240 GB budget and a 30-minute statement timeout. Database time then dominates the agents’ time: it is 97.8% of the median attempt’s end-to-end time, against 49.0% at scale factor 10, and a run without AXR spends 27,647 s in the database. AXR saves 23.9% of it (paired 21.2–25.6) and C1i 21.6%; no statement timed out, and AXR wrote 29 GB per run. No wait for an in-flight result lasted over 88.9 s at scale factor 10; here, 37 of C6’s 178 waits

(and 28 of C1i’s) reached the 300 s limit, after which the waiter computes the result itself (§3.4). These waits cost 13.0–14.1 points of C0’s database time per C6 run and 9.7–11.0 per C1i run; had the waiters computed at once, C6 would have saved about 37% and C1i 32% (an estimate that ignores the added contention). Where the statement’s text identifies the builder (27 cases), it finished 51–367 s after the timeout, in 22 cases sooner than the waiter’s own recomputation. This suggests letting a wait last as long as the builder makes progress, unless recomputing is cheaper, rather than a fixed time. As the think times come from scale factor 10 and N4 has recall, the database share bounds this replay rather than describing live agents, whose attempts would also overlap differently.

More parallel workers. With up to 16 parallel workers in total instead of 8, so that four concurrent statements can each use four, a run of N4 without AXR spends 700 s instead of 881 s in the database. AXR then saves 38.9% and C1i 32.4% (49.1% and 44.3% with 8 workers). On DS4b, AXR saves 23.7% and C1i 7.5% (29.5% and 8.2%). The relative saving shrinks, but subtree reuse keeps most of its advantage on DS4b ($\Delta = 16.3$ against 21.3 points).

5.7 Overheads

Latency of statements that do not benefit. We compare each statement of at least 0.1 s with the same statement in the C0 run of the same block. For statements in which AXR neither stores nor reuses anything, the median latency ratio is 0.98–1.00 across all twelve workloads, and the 99th percentile 1.23–2.61; for the statements that C1i leaves untouched, the 99th percentile is 1.14–3.08, of the same order, so these tails reflect variation between runs more than an overhead of AXR. Statements that store a result without reusing one have a median ratio of 1.00–1.04 and a 99th percentile of up to 3.51: writing a large result can slow the statement that builds it.

Storage and read-back. AXR writes 3.1 GB per run on N4 and up to 55.4 GB on TPC-DS (Table 3), against a 24 GB budget. A micro-benchmark that stores a filtered scan of `lineitem` (about 40 MB to 4.8 GB) and reads it back through a Reuse node measured 0.71–0.75 ms per MB, likely from the page cache; a cold sequential read with direct I/O from the NVMe RAID ran at 5.6 GB/s. Both are well below the 1 ms per MB that cost-based admission assumes (§3.5), so the assumption errs on the side of admitting too little.

Comparing descriptions. With the final build (§4), C0 and C6 on N4 and DS4b (three blocks each, server of Table 4) gave C6 reductions of 50.9% and 27.6% (Table 4: 47.6% and 29.2%). Its 916 comparisons took at most 0.024 ms each, under 0.0005% of database time, and found no difference, so the results of the evaluation build carry over. The reductions differ by more than the paired ranges because run times shift between experiments run on different days, which pairing within one experiment does not capture: C0 took 945–1,006 s on N4 here against 890–961 s for Table 4, and 2,701–2,751 s against 2,597–2,660 s on DS4b.

5.8 Correctness

AXR’s correctness rests on its design rules (§3.2, §3.6) and on targeted tests with assertion-enabled builds (§4); the checks of this section are regression checks on top. Across all concurrent

Table 4: Ablation (§5.5; second server): reduction vs. C0 (% , mean of three runs). Step: change from the row above in points of C0 database time, with the paired minimum and maximum (C2 differs from C1 in granularity and coverage; each later row adds one element). Mk: makespan reduction; GB: data written per run.

Config	TPC-H N4				TPC-H N16				TPC-DS DS4b				TPC-DS DS4bR2				TPC-DS DS4bPro								
	DB	Step	Mk	GB	DB	Step	Mk	GB	DB	Step	Mk	GB	DB	Step	Mk	GB	DB	Step	Mk	GB					
C1 stmt	24.1	–		20.7	0.0	41.2	–		3.0	0.0	7.3	–		1.7	0.5	11.4	–		11.4	0.5	11.7	–		2.9	0.0
C2 exact cols	14.4	–9.7	[–11.4, –7.8]	14.1	14.0	30.4	–10.7	[–12.6, –9.7]	2.8	1.3	3.5	–3.8	[–4.5, –2.7]	2.1	37.5	4.0	–7.4	[–9.3, –5.0]	1.5	38.3	6.8	–4.9	[–6.8, –2.9]	2.3	18.4
C3 +col. subset	10.3	–4.1	[–6.4, –1.6]	8.4	14.0	30.9	0.5	[0.2, 0.8]	3.1	1.3	3.0	–0.5	[–0.7, –0.2]	0.5	37.5	6.4	2.3	[–4.1, 9.2]	3.6	33.3	7.7	0.9	[0.3, 1.8]	2.4	16.5
C4 +coverage	20.8	10.5	[10.0, 11.5]	17.2	15.8	44.5	13.5	[12.1, 15.4]	5.7	13.6	22.2	19.2	[18.5, 20.5]	9.8	77.8	32.8	26.4	[22.7, 30.4]	21.0	76.2	34.0	26.3	[25.1, 28.1]	16.7	41.9
C5 +admission	27.7	6.9	[3.7, 9.9]	21.9	3.4	45.2	0.7	[0.2, 1.2]	7.3	5.6	28.0	5.8	[4.8, 6.3]	15.2	46.7	34.3	1.5	[–1.5, 3.1]	22.0	56.1	34.8	0.8	[0.0, 1.5]	16.5	38.0
C6 AXR	47.6	19.9	[16.6, 22.5]	37.5	3.1	51.4	6.2	[5.8, 6.5]	13.3	5.5	29.2	1.2	[0.3, 2.8]	16.4	46.7	35.4	1.1	[–1.1, 3.6]	23.0	55.4	36.5	1.7	[0.6, 3.1]	16.8	37.9

experiments, 724,011 statement executions with AXR enabled were checked against the results of the same statements without AXR, as were the 52,103 statements of 56 sequential correctness replays. These counts include runs on development builds, and on their own they are weak evidence: the look-alike column bug of §3.2 survived 68,790 checked executions and was found by a targeted test. No result differed except for the EXPLAIN statements of the renamed TPC-H workloads, whose plan descriptions show AXR’s nodes by design, and statements whose LIMIT does not follow an ORDER BY, follows ORDER BY random() or falls between tied rows, where several answers are correct. In every workload and configuration of Table 3 but DS4c and DS4cR2, a run with AXR differed on no more of these statements than a run without AXR differs from the other runs without it. On those two, runs with reuse differed on up to 4 such statements on which the runs without AXR agreed. Re-executing the ordered ones showed ties in both. On DS4c, 62 groups with a NULL total sort first, so any 10 of them answer the LIMIT 10, and a C6 replay of the attempt returned 10 of them. On DS4cR2, PostgreSQL reduces the sort of a GROUPING SETS query to one column, which leaves two rows tied, and PostgreSQL without AXR returned both answers.

Two further checks target what the replays may miss. First, a generator of random TPC-H queries built from shared blocks (foreign-key joins, filters with few literal values, grouping, IN, EXISTS and scalar sub-queries, common table expressions, window functions and ordered LIMITs), so that subtrees recur across queries, produced 1,958 queries that finished within 60 s on scale factor 1. On an assertion-enabled build, each ran once without and twice with AXR, with every result admitted and results compared exactly, value by value: 376 queries reused a stored result in their first run with AXR and all of them in their second, and no result differed. Second, in sequential replays of N4 and H4o (561 and 744 statements) with C1 and C5 settings, we compared every result exactly, without rounding: no result differed.

6 DISCUSSION AND LIMITATIONS

Regime. AXR pays off when database time is a large part of an agent’s time. We evaluated it on PostgreSQL at scale factors 10 and 100, where that holds. On small databases it does not, and on a fast columnar engine the database share of the same statements is small at scale factor 10 but similar to PostgreSQL’s at scale factor 100 (§2.3). We did not implement AXR for such an engine, so whether reuse pays off there is open. There, recomputing a result is cheaper

relative to reading it back, so the admission rule of §3.5 would reject more results. Our sequential event logs suggest not many: if every stored subtree were built 18.7 times faster while reading it back cost the same, the rule would still keep 81–92% of the results and 76–100% of the saving from their reuse, because most stored results are small compared with the work that produced them. On such an engine, the obstacle at scale factor 10 is how little database time there is to save, not admission. For an operator, AXR reduces latency and database compute, not the cost of LLM calls, which replay leaves unchanged. The database share, and with it the end-to-end benefit, falls with slower LLMs (Table 1; enabling the pro model’s reasoning mode would likely lower it further) and rises with the data (§5.6).

Agents and workloads. All traces come from one agent design and two LLMs of one provider, and the questions derive from public benchmarks. Public queries invite recall: on TPC-H the agents reproduced the reference queries, so their attempts converge more than attempts at new questions would (§5.2). Renaming removes recall of names but not necessarily of query structure, and fully specified questions still make attempts converge; questions of real users over private schemas remain untested. The benefit on TPC-H depends on running several attempts per question at once: almost all of its reuse is across attempts (Fig. 1), and a statement cache that shares running statements obtains most of it. An agent that runs a single attempt would see mostly the reuse within an attempt, which subtree reuse serves (§5.3). The TPC-DS questions are hard for the agent (7.5–13.3% accuracy), so these traces describe agents that struggle. The three TPC-DS question sets show that the benefit varies with the questions (AXR saves 6.5%, 13.1% and 29.5% on DS4c, DS4 and DS4b), and the gain of subtree reuse comes from a few of them (§5.3): it was large on every recording of one set and small on two of the three recordings of another.

Replay instead of live agents. We replay recorded traces with their original think times. In a live system, faster answers could change what an agent does next, and attempts would finish earlier and overlap differently. Replay isolates the database effect on identical statements but does not capture this feedback. A closed-loop replay measures part of it without an LLM: four workers take the attempts in recorded order, each starting the next as soon as one finishes. We ran it on N4, N16 and DS4b with the final build (C0, C1i and C6, three blocks); apart from LIMITs without a deterministic order, no result differed. The closed loop compresses the workloads: with C6, the span from first to last statement shrinks

by 18.7–34.1% instead of 0.2–7.2%, and the mean time from a question’s first attempt to its last by 17.6–40.7% instead of 6.1–34.6%. Database time changes less (C6 30.5–49.6% against 30.3–51.0%; Δ 4.4–20.5 against 3.8–20.9 points), most on N4 (C6 50.8% to 44.0%), whose attempts then overlap more: C6 waits for results being built 30 instead of 20 times per run.

Concurrency. Concurrent attempts contend for the server (§5.4). AXR saves work but does not schedule it; coordinating when sibling attempts run their expensive statements could reduce contention further. Waiting for a result that is still being built has a fixed timeout, which is too short at scale factor 100 (§5.6).

Writes and replication. AXR is correct under concurrent writes executed through its instance (§3.6), but any write clears the cache and disables reuse while it is open, for every session. This suits analytical databases loaded in batches; the recorded agents never wrote. Agents that keep intermediate results in temporary tables would clear the cache with every CREATE; exempting temporary relations and invalidating per relation would avoid that and would also be needed for continuous writes. Changes that arrive through replication are not detected, so AXR does not support read replicas.

Deployment and access control. AXR must be loaded at server start (`shared_preload_libraries`), which requires a self-managed PostgreSQL; many managed services do not allow it. Results are shared by all sessions, but every statement is still subject to PostgreSQL’s permission checks on the relations it reads, because AXR rewrites plan nodes but not the statement’s range table; we have not evaluated several roles or row-level security.

Policy. Cost-based admission with LRU eviction cut the data written per run on N4 from 15.8 GB to 3.4 GB (§5.5), but on TPC-DS AXR still writes about twice its result budget per run (§5.7) and evicts results while the workload runs. A policy derived from traces did not help: a size cap with frequency-aware GreedyDual-Size eviction [6], tuned in simulation on DS4, wrote a third of the data on DS4 but saved 21.9% instead of 36.6% on the held-out DS4b (on the earlier server of §5.5), because the largest intermediate results were among the most reused, so we kept cost-based admission and LRU. A policy that predicts reuse from agent behavior, e.g., from the structure of an attempt’s statements (§5.3), remains open.

7 RELATED WORK

Data systems for agents. Liu et al. [21] argue that data systems should be redesigned for LLM agents, whose *agentic speculation* (high-volume, heterogeneous, redundant and steerable exploratory queries) differs from human workloads; they report that across 50 parallel attempts per BIRD text-to-SQL task, distinct sub-plans are often less than 10–20% of the total. AXR is a concrete instance of that agenda inside an existing engine, and our measurements refine its premise: on small databases database time is negligible (§2.3), and the form of the redundancy differs between workloads (§2.4). Halo [30] and Helium [31] optimize agentic workflows as query plans over LLM and tool calls and reuse work across calls; they act above the database. GraftDB [18] is the closest work: a multi-query engine that folds later-arriving

Table 5: AXR’s mechanisms against the closest prior work.

Mechanism	Closest prior work	AXR
Matching	plan signatures [16]; view matching [9]	resolved signatures of physical subtrees, run-time parameters bound; column subsets; no subsumption
Unit of reuse	operator results [15]; hash tables [7]	any plan subtree, including parameterized and correlated fragments
Concurrency	shared operators and scans [5, 8, 14]; request coalescing [23]	wait for a result that is being materialized
Admission	cost/benefit of intermediates [11, 15]	build time against measured read-back time
Consistency	invalidation in result caches	writer registration and snapshot visibility; whole-cache clear

analytical queries into running executions and reuses operator state, motivated by agents that issue and refine queries. AXR differs in three respects: it extends an unmodified PostgreSQL instead of a new engine; it reuses materialized results across the think time that separates most agent queries, and shares in-flight results only by waiting for them; and it is evaluated on recorded LLM-agent traces rather than synthetic arrivals. Chronos [34] and BranchBench [2] address branching of writable state for speculative agents, a different problem; Kim and Ailamaki [17] envision research agents that avoid repeating work.

Reusing intermediate results. Multi-query optimization finds and materializes common subexpressions within a batch of queries [27, 29]. Materialized-view selection chooses results to keep for a known workload [1, 10, 13], and view matching answers a query from a materialized view whose definition subsumes the needed computation, including views with more columns or weaker predicates than needed [9, 12]. Recycling reuses intermediates of earlier queries opportunistically: in a column store’s operator-at-a-time engine [15], in a pipelined engine [22], as reusable hash tables [7], and with cost-based decisions in dataflow programs [11]. CloudViews [16] identifies common subexpressions of recurring production jobs by plan signatures. Table 5 places AXR’s mechanisms against the closest of these. AXR’s matching is narrower than view matching: it compares physical subtrees after optimization, accepts a stored result only if the signatures are equal up to the set of output columns, and does not reason about predicate subsumption, so it reuses only work that the chosen plans happen to share. What it adds is signing the fragments that dominate agent queries in PostgreSQL, correlated sub-plans and inner sides of parameterized nested loops, by binding their run-time parameters (§3.3).

Shared execution. QPipe [14], SharedDB [8], cooperative scans [35], CJOIN [5] and DataPath [3] share operators, scans and joins among concurrent queries. AXR’s in-flight waiting is a coarse form of this sharing: a waiter reads a result materialized anyway, once complete.

Result and semantic caches. Statement-level caches return a stored result for a repeated statement. Semantic caching answers

queries from cached results described by predicates, including partial overlaps [26]; LLM-based canonicalization extends statement caches to statements that are written differently but mean the same [4], and verified semantic caching reuses LLM responses for similar prompts [28]. AXR is exact and operates below the statement level; our statement-cache baselines (C1, and C1i with in-flight sharing) share AXR’s plan-based matching, which recognizes many statements whose text differs (§5.2). For whole statements, in-flight sharing is request coalescing, as in memcache’s leases [23]; AXR coalesces plan subtrees inside the executor, which requires the deadlock argument of §3.4. We did not compare with GraftDB [18], a separate engine, or with materialized views, which must be defined in advance.

8 CONCLUSION

LLM agents that explore large databases repeat their own and their siblings’ work. AXR reuses their intermediate results inside an unmodified PostgreSQL and reduced database time on all twelve recorded workloads. Where attempts converged, reusing whole statements and sharing them while they run obtained most of the saving. Reusing subtrees paid off on one held-out TPC-DS question set and much less on a second, so its value depends on the questions. Evaluations of agents on public benchmarks should check for recall, and evaluations of reuse should compare against a statement cache that shares running statements, not only against no reuse.

REFERENCES

- [1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. 2001. Materialized view and index selection tool for Microsoft SQL server 2000. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. ACM, 608. <https://doi.org/10.1145/375663.375769>
- [2] Elaine Ang, Sam Weldon, In Keun Kim, Kevin Durand, Kostis Kaffes, and Eugene Wu. 2026. BranchBench: Aligning Database Branching with Agentic Demands. arXiv:2604.17180
- [3] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 519–530. <https://doi.org/10.1145/1807167.1807224>
- [4] Laurent Bindschaedler. 2026. Semantic Caching for OLAP via LLM-Based Query Canonicalization (Extended Version). arXiv:2602.19811
- [5] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment* 2, 1 (Aug. 2009), 277–288. <https://doi.org/10.14778/1687627.1687659>
- [6] Ludmila Cherkasova and Gianfranco Ciardo. 2001. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *High-Performance Computing and Networking*. Springer Berlin Heidelberg, 114–123. https://doi.org/10.1007/3-540-48228-8_12
- [7] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1275–1289. <https://doi.org/10.1145/3035918.3035957>
- [8] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment* 5, 6 (Feb. 2012), 526–537. <https://doi.org/10.14778/2168651.2168654>
- [9] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. ACM, 331–342. <https://doi.org/10.1145/375663.375706>
- [10] Himanshu Gupta. 1997. Selection of views to materialize in a data warehouse. In *Database Theory — ICDT ’97*. Springer Berlin Heidelberg, 98–112. https://doi.org/10.1007/3-540-62222-5_39
- [11] Stefan Hagedorn and Kai-Uwe Sattler. 2018. Cost-Based Sharing and Recycling of (Intermediate) Results in Dataflow Programs. In *Advances in Databases and Information Systems*. Springer International Publishing, 185–199. https://doi.org/10.1007/978-3-319-98398-1_13
- [12] Alon Y. Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal* 10, 4 (Dec. 2001), 270–294. <https://doi.org/10.1007/s007780100054>
- [13] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD ’96*. ACM Press, 205–216. <https://doi.org/10.1145/233269.233333>
- [14] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastasia Ailamaki. 2005. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 383–394. <https://doi.org/10.1145/1066157.1066201>
- [15] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. 2009. An architecture for recycling intermediates in a column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 309–320. <https://doi.org/10.1145/1559845.1559879>
- [16] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifeng Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 191–203. <https://doi.org/10.1145/3183713.3190656>
- [17] Kyounghmin Kim and Anastasia Ailamaki. 2026. Confining Nondeterminism: AI-Driven Research Systems as DBMSs for Reliable, Non-Wasteful, Transparent, and Collaborative Research [Vision]. arXiv:2607.10508
- [18] Genki Kimura and Kazuo Goda. 2026. GraftDB: Dynamic Folding of Concurrent Analytical Queries. arXiv:2606.04303
- [19] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2024. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. arXiv:2411.07763
- [20] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. arXiv:2305.03111
- [21] Shu Liu, Soujanya Ponnappalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. arXiv:2509.00997
- [22] F. Nagel, P. Boncz, and S. D. Viglas. 2013. Recycling in pipelined query evaluation. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 338–349. <https://doi.org/10.1109/icde.2013.6544837>
- [23] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, 385–398.
- [24] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. arXiv:2410.01943
- [25] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 1981–1984. <https://doi.org/10.1145/3299869.3320212>
- [26] Qun Ren, M.H. Dunham, and V. Kumar. 2003. Semantic caching and query processing. *IEEE Transactions on Knowledge and Data Engineering* 15, 1 (Jan. 2003), 192–210. <https://doi.org/10.1109/tkde.2003.1161590>
- [27] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhohe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. ACM, 249–260. <https://doi.org/10.1145/342009.335419>
- [28] Luis Gaspar Schroeder, Aditya Desai, Alejandro Cuadron, Kyle Chu, Shu Liu, Mark Zhao, Stephan Krusche, Alfons Kemper, Matei Zaharia, and Joseph E. Gonzalez. 2025. vCache: Verified Semantic Prompt Caching. arXiv:2502.03771
- [29] Timos K. Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems* 13, 1 (March 1988), 23–52. <https://doi.org/10.1145/42201.42203>
- [30] Junyi Shen, Noppanat Wadlorn, and Yao Lu. 2025. Batch Query Processing and Optimization for Agentic Workflows. arXiv:2509.02121
- [31] Noppanat Wadlorn, Junyi Shen, and Yao Lu. 2026. Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective. *Proceedings of the ACM on Management of Data* 4, 3 (May 2026), 1–29. <https://doi.org/10.1145/3802046>
- [32] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171
- [33] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language

- Models. arXiv:2210.03629
- [34] Xinjing Zhou, Jason Mohoney, Samuel Madden, Michael Stonebraker, and Lei Cao. 2026. Chronos: Efficient Bolt-on Branching Across Data Stores for Stateful Agentic Applications. arXiv:2609.14889
- [35] Michał Świtakowski, Peter Boncz, and Marcin Zukowski. 2012. From cooperative scans to predictive buffer management. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1759–1770. <https://doi.org/10.14778/2367502.2367515>