Materialize

WHITEPAPER

Materialize vs Clickhouse

Modern applications and AI systems depend on context.

They operate on business objects such as customers, orders, and accounts. These customers, orders and accounts are themselves assembled from operational tables joined together, rules applied, and values aggregated across systems. These business objects are derived state: when an input changes, the object must change with it.

Maintaining that state requires more than a database optimized for storing rows or scanning history. It requires a system that continuously assembles and maintains the current shape of the business as data changes. This is a **context engine** for the business.The modern business runs on “derived state.”

Context engines and live data products

A context engine system produces live data products, which are derived datasets built from multiple sources, kept current as those sources change, and served directly to systems that act on them. Applications display them. APIs expose them. Automation workflows and AI agents make decisions from them.

A context engine serving live data products demands three properties. The data must be fresh and reflect what is true now. It must be correct, preserving updates, deletes, and transactional boundaries so results never reflect partial state. It must also be composable, allowing derived views to build on one another without introducing timing gaps or stale intermediate layers.

These requirements reinforce each other. Fresh but incorrect data accelerates mistakes. Correct but stale data misleads downstream systems. Composable but inconsistent data spreads errors.

AI agents make this constraint unavoidable. An agent that updates an order, checks inventory, and recalculates fulfillment performs dependent reads and writes in sequence. Each step assumes the last has taken effect. High concurrency and multi-step workflows leave little tolerance for delay or inconsistency.

This paper compares two systems through that lens: Materialize and ClickHouse. Both integrate with operational databases and support modern data-driven applications. Their designs, however, reflect different views of what the data layer should provide to the business.

Materialize is built to continuously maintain the current shape of the business as data changes.

It keeps customer records, orders, accounts, and other core entities aligned across systems so that applications, automation, and AI workflows operate on a consistent and up-to-date representation of the organization.

ClickHouse is built to execute fast queries over large volumes of stored data.

It is optimized for summarizing activity, exploring trends, and analyzing historical information at scale.

These architectural priorities lead to different strengths. One approach centers on keeping operational state coherent and ready for action. The other centers on delivering high-performance analysis across accumulated data.

For teams investing in customer-facing features, automation, and AI-driven workflows, this distinction shapes what the data platform can enable. The choice determines whether the system primarily serves as a foundation for ongoing business operations or as a high-performance engine for large-scale analysis.

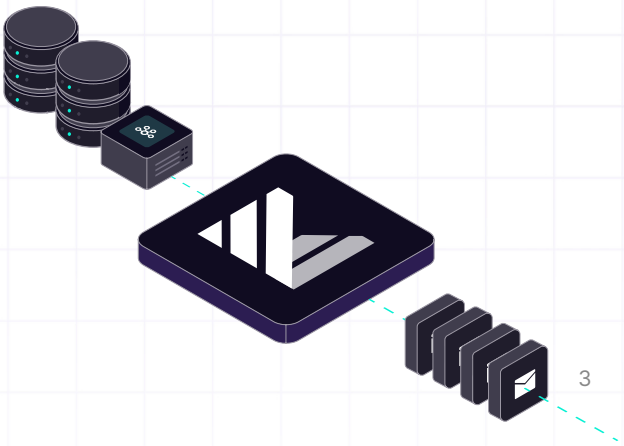
A context engine runs on live data products

A context engine system produces **live data products:** derived datasets built from multiple sources, kept current as those sources change, and served directly to systems that act on them. Applications display these live data products. APIs expose them, and automation workflows and AI agents make decisions from them.

Serving live data products demands three properties. The data must be fresh and reflect what is true now. It must be correct, preserving updates, deletes, and transactional boundaries so results never reflect partial state. It must also be composable, allowing derived views to build on one another without introducing timing gaps or stale intermediate layers.

These requirements intersect and reinforce each other. Data that is fresh but not correct leads to mistakes. Data that's correct but stale makes downstream systems go astray. And data that is composable but inconsistent spreads errors throughout any views depending on it.

AI agents make this constraint unavoidable, however. An agent that updates an order, checks inventory, and recalculates fulfillment performs dependent reads and writes in sequence. Each step assumes the last has taken effect. High concurrency and multi-step workflows leave little tolerance for delay or inconsistency.



This paper compares two systems, Materialize and ClickHouse, through the lens of data freshness, correctness, and composability. Both integrate with operational databases and support modern data-driven applications. Their underlying architectures, however, reflect different views of what the data layer should provide to the business.

- **Materialize** is built to continuously maintain the current shape of the business as data changes. It keeps customer records, orders, accounts, and other core entities aligned across systems so that applications, automation, and AI workflows operate on a consistent and up-to-date representation of the organization.
- **ClickHouse** is built to execute fast queries over large volumes of stored data. It is optimized for summarizing activity, exploring trends, and analyzing historical information at scale.

These design priorities lead to different strengths. The Materialize approach centers on keeping operational state coherent and ready for action. The ClickHouse approach centers on delivering high-performance analysis across accumulated data.

For teams investing in customer-facing features, automation, and AI-driven workflows, this distinction shapes what each of these data platforms can enable. The choice determines whether the system primarily serves as a foundation for continual business operations or as a high-performance engine for large-scale historical data analysis.

Operational Systems + Reaction Time: Data freshness, correctness, and composability

Operational systems don't just store data, they act on data. Operational systems exist to respond to change, and they depend on freshness, correctness, and composability working together to minimize reaction time, which is the delay between a real world event and the system's response to it.

Freshness

Reaction time begins with visibility. An application rendering account details, an API returning order status, or an agent applying a policy assumes that the context it's given reflects current state. Data that lags behind events extends reaction time because the organization is now acting on outdated information.

Correctness

Reaction time also depends on confidence. Customers update profiles, orders change status, balances adjust, and **changes committed together must become visible together.** If derived results reflect partial or conflicting updates, downstream systems hesitate or require compensating logic. Confidence declines, and reaction slows.

Composability

Business entities span systems. A customer's eligibility may depend on order history, account standing, and profile data maintained in three separate locations. If these derived entities cannot be composed reliably, every new workflow introduces an additional coordination and synchronization burden. Each added layer increases latency between event and decision.

Minimizing reaction time requires simultaneous freshness, correctness, and composability. Speed without confidence does not shorten reaction time. Confidence without freshness does not shorten reaction time. Composability without either introduces coordination delays that offset gains.

Architecture dictates data change priorities

When data changes, systems can respond in two fundamentally different ways:

- **Change as primary.** Mutations are ingested as they occur. Derived state is maintained incrementally. Consistency is preserved as a system-wide property.
- **Stored data as primary.** Data is appended or written in batches. Queries reconcile versions at read time. Updates and deletes are handled through background processes and deduplication.

Each approach reflects a different workload priority.

Materialize is designed as a context engine for operational workloads.

It connects directly to systems such as PostgreSQL, MySQL, SQL Server, and Kafka through change data capture or native connectors. Ingestion, computation, storage, and serving operate on a shared logical timeline. Derived data products are defined in standard SQL and maintained incrementally as source data changes.

ClickHouse is designed as a high-performance columnar database optimized for scanning and aggregating large datasets.

Its architecture emphasizes compression, vectorized execution, and efficient storage for substantial volumes of data. It excels at analyzing historical activity and summarizing trends across accumulated datasets.

ClickHouse's benchmark strategy illustrates its stored-data-as-primary approach.

Its primary public benchmark, ClickBench, evaluates performance by running aggregate queries against a single large table of roughly one hundred million rows. The workload consists of sequential analytical queries without joins across multiple normalized tables, without updates or deletes, and without concurrent user activity. It measures scan speed and aggregation throughput.

ClickBench demonstrates the strengths ClickHouse is designed for: large-scale analytical processing across stored data. It does not measure incremental maintenance of derived state, cross-table transactional consistency, or performance under concurrent operational workloads.

Benchmarks reveal priorities, and ClickBench reflects an architecture optimized for analytical throughput, not data freshness.

Freshness in practice

The ability to act on current rather than prior state depends on how quickly changes move from source systems into the derived views that applications and agents actually read.

How Materialize handles data freshness

In Materialize, changes are processed as they arrive.

- When a row changes in a source PostgreSQL database, the change is ingested through a direct CDC connection. All affected views update incrementally, including views built on other views. Results become available through the same continuous pipeline.
- Because derived state is maintained ahead of time, query latency remains stable even as underlying data grows. Applications and services read from maintained state rather than recomputing it on demand.

How ClickHouse handles data freshness

ClickHouse ingests Postgres changes through ClickPipes on a configurable interval, with a default measured in minutes.

- This ingestion interval introduces a latency floor between source systems and derived results. Even at lower intervals, data in ClickHouse reflects a prior state of the source.
- For operational systems that act immediately on what they read, this delay shapes system behavior. As workflows chain reads and writes, timing gaps accumulate, propagating and growing through each downstream dependency.

Correctness in practice

Holding confidence that derived state reflects complete, consistent updates is essential to reducing reaction time. How a system maintains that correctness under continuous mutation determines whether downstream consumers can trust those inputs without additional safeguards.

How Materialize handles mutable data

Materialize assigns each change a position on a logical timeline and evaluates queries against consistent points on that timeline.

- Views advance together as their inputs advance. Results become visible only when all contributing data has reached the same logical position.
- Transactional boundaries are preserved. Changes committed together upstream become visible together downstream, and queries that join multiple tables observe a consistent snapshot without additional annotations.

In workloads where many sessions issue similar lookups or entity-level reads, this repeated reconciliation becomes a primary contributor to latency and resource consumption.

CPU and memory are spent consolidating historical versions rather than evaluating business logic. The cost scales with concurrency rather than remaining amortized across the system.

How ClickHouse handles mutable data

ClickHouse handles mutable data through append and merge.

- Updates and deletes replicated through CDC are written as new rows in ReplacingMergeTree tables. Background merges reconcile versions by primary key and retain the most recent version.
- This design supports high-throughput ingestion and efficient analytical storage. Until background merges complete, multiple versions of a row may remain visible. Queries can apply the FINAL keyword to force version consolidation before execution. In normalized schemas, this must be applied across each table participating in a join.
- When FINAL is used, ClickHouse performs reconciliation work as part of query execution. Each table must be read, sorted by its primary key, and reduced to the latest version before the join can proceed. This work is not shared across concurrent queries. As query volume increases, reconciliation cost increases proportionally because each query repeats the same consolidation steps.

Composability in practice

Composability — the ability to build complex business entities from simpler derived components without introducing coordination overhead — depends on how a system maintains relationships across layered views.

How Materialize handles composability and materialized views

In Materialize, views compose because they share a timeline and are maintained incrementally.

- A view can depend on another view without introducing refresh scheduling or synchronization logic. As base data changes, each layer updates within the same consistent pipeline.
- This creates a maintained dependency graph of derived state that remains internally consistent. Applications and AI systems query this graph directly.

How ClickHouse handles composability and materialized views

ClickHouse supports two forms of materialized views:

- Continuous materialized views act as insert triggers. When rows are inserted into a source table, the view processes those inserted rows within the same batch. This model is effective for pre-aggregating append-only data. It does not maintain global consistency across existing rows in a table and does not automatically respond to updates in joined dimension tables.
- Periodic materialized views perform scheduled recomputation. At defined intervals, the system re-executes a query and replaces the target table. This aligns with reporting workloads where freshness is measured in minutes or hours.

Both of these materialized view forms reflect ClickHouse's primary orientation toward analytical workloads. They do not provide incremental, transactionally consistent maintenance of composed business entities that span multiple mutable tables.

AI workloads as a multiplier

AI agents amplify the architectural distinction between Materialize and ClickHouse.

In **Materialize,** agents connect through the PostgreSQL interface or through an MCP server and read from continually maintained state.

Scaling concurrent sessions does not multiply recomputation cost because derived results are always kept current.

In **ClickHouse,** reconciliation work occurs at query time.

As concurrent agent sessions increase, reconciliation overhead scales with them. Cross-table consistency remains dependent on merge timing and query structure.

As AI-driven systems become centralized within business operations, maintaining coherent, up-to-date operational state becomes foundational infrastructure.

Two data systems, two different objectives

Materialize and ClickHouse are both powerful data systems. They are optimized for different objectives.

ClickHouse delivers high-performance analytical processing across large datasets and supports exploration and reporting at scale.

Materialize is designed to minimize reaction time by continuously maintaining derived operational state as source data changes. It supports applications, automation systems, and AI agents that depend on a coherent and current representation of the business. It is the ideal architecture for organizations building responsive products and AI-driven workflows, reducing reaction time becomes a strategic advantage. Architecture determines how quickly the business can move from event to confident decision.