Virtualizing Cloud Data Infrastructures with BRAD

Geoffrey X. Yu
MIT CSAIL
Cambridge, MA, USA
geoffxy@mit.edu

Ziniu Wu
MIT CSAIL
Cambridge, MA, USA
ziniuw@mit.edu

Ferdi Kossmann
MIT CSAIL
Cambridge, MA, USA
kossmann@mit.edu

Tianyu Li
MIT CSAIL
Cambridge, MA, USA
litianyu@mit.edu

Markos Markakis
MIT CSAIL
Cambridge, MA, USA
markakis@mit.edu

Amadou Ngom
MIT CSAIL
Cambridge, MA, USA
ngom@mit.edu

Sophie Zhang
MIT CSAIL
Cambridge, MA, USA
sophiez@mit.edu

Tim Kraska
MIT CSAIL, AWS
Cambridge, MA, USA
kraska@mit.edu

Samuel Madden
MIT CSAIL
Cambridge, MA, USA
madden@csail.mit.edu

Abstract

Organizations usually manage their data using multiple specialized cloud database engines (e.g., Aurora, BigQuery, etc.). However, designing and managing multi-engine infrastructures is hard; there can be many designs, each with different performance and costs. Changing the design afterwards (e.g., due to growth) is even more challenging since application code usually ends up tightly coupled to the engines. We propose data infrastructure *virtualization*. The key idea is to declare a set of *virtual database engines* (VDBEs), which specify an engine's application-facing properties (e.g., query interface, performance) and its tables, but do not prescribe a concrete engine. An automated planner then decides how to best realize the VDBEs onto physical engines based on the workload. Clients connect to VDBE endpoints and are oblivious to the underlying physical engines—allowing for seamless infrastructure changes. We implemented VDBEs and an automated planner in Brad: the first data infrastructure virtualization runtime. Our demo will showcase VDBEs and Brad's automated planner under different workloads.

CCS Concepts

• **Information systems → Data management systems**.

Keywords

Cloud Databases; Data Infrastructure; Virtualization; HTAP

ACM Reference Format:
Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Sophie Zhang, Tim Kraska, and Samuel Madden. 2025. Virtualizing Cloud Data Infrastructures with BRAD. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725141

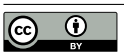
1 Introduction

Organizations typically use a collection of specialized cloud database engines (e.g., Aurora, Redshift, BigTable, etc.) to handle their data needs, each chosen for the performance, functionality, and cost benefits they bring to different parts of their workload. But these multi-engine infrastructures are notoriously difficult to design and manage. To pick the "right" engine(s) to use, engineers must choose among dozens of specialized database engines, all with different performance and costs [5]. For example, suppose that a company wants to make ticket sales while also running advanced sales analytics on the same data. They could use a single DBMS (e.g., PostgreSQL) to serve both data use cases, leverage a specialized HTAP engine (e.g., TiDB), or choose a classical OLTP and OLAP engine setup (e.g., Aurora with Redshift). The right design depends on their scale (e.g., hundreds vs. millions of users), budget, and data freshness needs [9]. Regardless of their choice, changing the engines later on (e.g., because of a design mistake or growth) is even harder as they might have application code relying on the specifics of the chosen engines (e.g., using Redshift-specific APIs).

We propose a new declarative and automated approach as an alternative to manual infrastructure design. The key idea is to *virtualize* the data infrastructure. Developers declare a set of *virtual database engines* (VDBEs), which specify an engine's application-facing properties (SQL dialect, performance) and its tables. An automated planner then decides how to best realize the VDBEs onto physical engines based on their workload and monetary budget. The planner is free to map each VDBE to its own physical engine, or place multiple VDBEs on the same physical engine—similar to how multiple virtual machines can run on the same physical server. It can also map VDBEs to engines that use a different SQL dialect, provided that the VDBE's dialect can be translated or emulated, which we discuss in Section 2.3. If the workload changes, the planner can likewise change the physical engines in response.

The key twist in the VDBE abstraction is that the same table can be in multiple VDBEs, as shown in Figure 1 Ⓐ. Each VDBE represents a logical snapshot of data; if a table appears in multiple VDBEs, it corresponds to a different logical snapshot in each. This property is powerful because it lets developers define different use cases over the same logical data. For example, our company from earlier could create one VDBE to support high-throughput low-latency queries for their ticket sales (representing a traditional "transactional DBMS") and another VDBE to handle complex long-running queries for sales reporting (representing a traditional "data warehouse"), both with the same set of tables. Queries and transactions are issued to specific VDBEs, so they will always run against transactionally-consistent snapshots of data.

Related logical snapshots (i.e., different VDBEs with overlapping table sets) can be mapped to the same set of physical tables



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725141

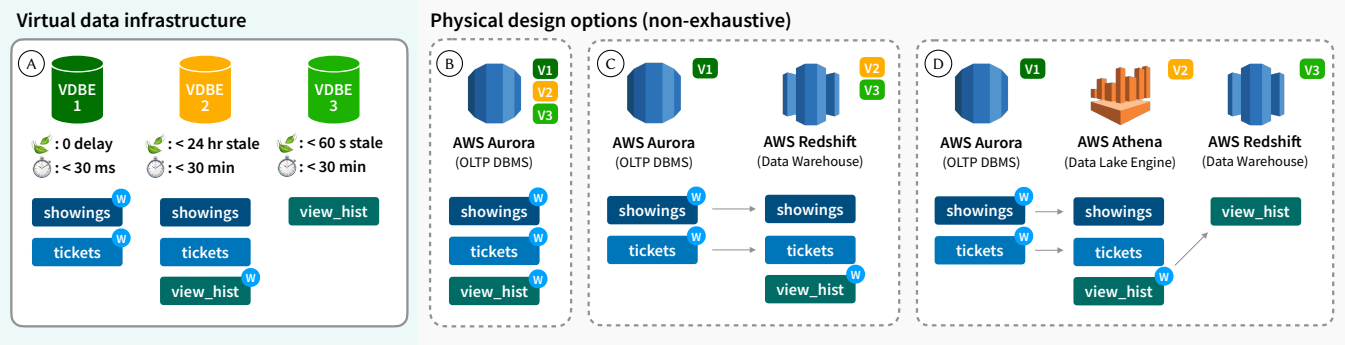


Figure 1: QuickFlix's virtual infrastructure alongside three possible physical designs. The "W" means the engine writes to the table and the gray arrows indicate the direction of physical data replication.

or to distinct physical replicas. Developers declare a data freshness constraint on each VDBE, which dictates the maximum staleness the VDBE can have with respect to writes made to its tables on any VDBE. The automated planner is responsible for realizing the logical snapshots in a physical infrastructure that meets the VDBEs' freshness constraints. For example, the planner could leverage cloud-native technologies like zero-ETL [2] to replicate data across physical replicas or rely on periodic data synchronization tasks. We discuss data consistency across VDBEs in Section 2.2.

VDBEs are a new abstraction that joins related work on automatically optimizing single systems and multi-engine infrastructures, which we discuss in our full papers [5, 9]. Unlike past approaches, VDBEs unify performance, cross-engine data freshness, and query dialects into a single abstraction for the cloud.

We have implemented VDBEs and an automated planner in Brad, the first data infrastructure virtualization runtime. Automatically realizing VDBEs is a challenging problem as it entails predicting query performance and costs over an exponentially large space of hypothetical designs; we detail how Brad's planner solves this problem in our full paper [9]. Operationally, Brad implements VDBEs using a database proxy [3]—exposing one endpoint per VDBE. Clients issue queries to a VDBE endpoint and the proxy then forwards the queries to a physical engine for execution.

Our demo showcases the VDBE abstraction and Brad's automated planner using an interactive dashboard (Figure 2). The dashboard visualizes the VDBEs, the physical infrastructure, and real-time performance metrics as a workload executes on Brad. To see the automated planner in action, attendees will get to modify an executing workload's intensity; Brad will then show how it would change the physical infrastructure in response. To show how VDBEs make physical design changes seamless, attendees will get to add/remove VDBEs and manually change their mapping to physical engines; they will be able to issue queries to the VDBEs without having to reconnect between physical design changes. Finally, to show that VDBEs work with existing data tools, attendees will get to run AWS Glue ETL jobs that read from and write to VDBEs.

2 Virtual Data Infrastructures

Now that we have presented an overview of virtualized data infrastructures, we go through an example of using VDBEs in a practical scenario to illustrate the details behind the abstraction.

Table 1: A summary of the properties of a VDBE.

Property	Example for VDBE 1 in Figure 1
Table set (w/ schema and write flags)	`showings` (write), `tickets` (write)
Query interface (dialect and features)	SQL-99, supports transactions
Performance SLOs	p90 query latency $\leq$ 30 milliseconds
Data freshness	No staleness permitted

2.1 Virtual Database Engines at QuickFlix

Consider the data needs of QuickFlix, a fictitious movie theater company. QuickFlix is modernizing their operations and wants to start selling movie tickets online. They also want to run analytics over their sales data (e.g., to select better movie showing times). For simplicity, suppose they model their data using three tables: (i) `showings` for movie showings, (ii) `tickets` for ticket orders, and (iii) `view_hist`, to hold a history each customer's movie views and is derived from the data in `showings` and `tickets`. QuickFlix needs to access `showings` and `tickets` using transactions (e.g., adding new showings and recording ticket orders). Furthermore, they will only use `view_hist` for their sales analysis.

Thinking in data use cases. With VDBEs, QuickFlix designs their infrastructure by thinking about their *data use cases*. They have three use cases: (i) selling tickets, (ii) running analytics on their sales data, and more subtly (iii) keeping their `view_hist` table up-to-date with new ticket orders. After identifying their use cases, QuickFlix creates a VDBE for each use case (as shown in Figure 1 (A)). A VDBE declaration comprises (i) a set of tables, with their schemas and a per-table flag indicating whether or not the VDBE writes to the table; (ii) a query interface; (iii) performance service level objectives (SLOs); and (iv) a data freshness constraint. We summarize these properties and give examples in Table 1.

To make ticket sales, QuickFlix creates a VDBE (1) that contains the `showings` and `tickets` tables, supports transactions, and has a low query latency. For sales analytics, they have a VDBE (3) with the `view_hist` table. They set a performance SLO of 30 minutes since they will run large exploratory queries. To maintain the `view_hist` table, they create a VDBE (2) that holds the `showings`, `tickets`, and `view_hist` tables; this VDBE runs queries that transform the data in `showings` and `tickets` into `view_hist` (we discuss data consistency next). They again set a performance SLO of 30 minutes

as the transformation queries will take time to run. By using VDBEs, QuickFlix focuses on carving out their high-level data use cases instead of being bogged down with physical implementation details.

2.2 Logical Snapshots and Data Consistency

Recall that the same table can appear in multiple VDBEs (Figure 1 Ⓐ); each VDBE represents a logical snapshot of data.

Freshness constraints. Since tables can be in multiple logical snapshots, VDBEs have a data freshness constraint. Concretely, a VDBE V's freshness constraint specifies, for every table T in V, the maximum staleness that V's logical snapshot is allowed to have with respect to writes made to T on *any* VDBE. This staleness can be zero, which is for classical transactional use cases (e.g., ticket sales). A non-zero value means bounded staleness, which is typical in existing infrastructures that rely on periodic ETLs.

We can now look at QuickFlix's freshness constraints. VDBE 1 has zero staleness as it is used for ticket sales, which cannot run on stale data. VDBE 2 runs transformations that update `view_hist` using `showings` and `tickets`. Since `view_hist` is used for historical analytics, QuickFlix is fine with it being up to one day stale. This means the transformations can run using versions of `showings` and `tickets` that are up to one day stale. So VDBE 2 has a maximum staleness of one day. VDBE 3 supports analytics on `view_hist`. QuickFlix wants to use the latest version of `view_hist` once updates complete on VDBE 2, but does not need it to be immediately accessible. So VDBE 3 declares a maximum staleness of one minute.

Data consistency. Brad provides data consistency by only allowing at most one physical writer engine per table. Thus, if multiple VDBEs declare that they write to the same logical table, Brad will map the VDBEs' logical snapshots to the same physical tables on the same physical engine. This requirement is not fundamental, as we could use (typically expensive) external coordination to ensure consistent writes to the same table by multiple writer engines. Similarly, if a VDBE V declares a zero staleness constraint, Brad will map V's logical snapshot to the physical engine that writes to its tables. Brad periodically replicates the latest writes to physical table replicas to meet the other VDBEs' freshness constraints.

2.3 Query Dialects

Lastly, VDBEs must declare a SQL query dialect. This is an important practical consideration as different physical engines use different dialects, which are typically not fully compatible. Knowing the dialect ensures that VDBEs are mapped to an infrastructure that is capable of supporting the dialects its applications need.

Importantly, specifying a dialect does not necessarily constrain the VDBE to a single physical engine. When it is possible to automatically translate queries and emulate any missing functionality (e.g., special SQL functions specific to a dialect) [7], the planner can map a VDBE to a physical engine that implements a different dialect. If the VDBE's dialect is only partially translatable, the planner can also realize the VDBE on two physical engines: one having a different dialect that supports the translatable queries, and one for the remaining untranslatable queries. These physical design options are beneficial if they bring better performance or a lower overall cost when compared to using a physical engine that directly uses the declared dialect.

2.4 Automated Infrastructure Planner

Once we have a set of VDBEs, the final step is to realize them on physical engines. Automatically selecting an optimal physical design is challenging. A planner must consider intertwined design decisions (e.g., engine selection and workload assignment) while optimizing for performance and cost on the user's workload. We present a solution to this problem in our research paper on Brad [9]. Here, we provide a brief sketch of Brad's automated planner.

The key idea behind Brad's planner is to cast VDBE realization as a constrained cost-based optimization over a space of physical infrastructure design plans called *blueprints*. A blueprint consists of (i) the physical engines and their provisioning, (ii) the engines on which to place (and replicate) tables, and (iii) a policy for routing queries (issued to a VDBE) to a physical engine for execution. VDBEs are constraints imposed on blueprints. For example:

- The chosen physical infrastructure must meet the VDBEs' SLOs (e.g., queries should meet the latency constraint if one is set).
- The VDBE must be mapped to a physical engine that supports its desired query interface (or Brad can emulate the interface [5]).
- If two VDBEs write to the same table, their tables must be mapped to the same physical replica on one physical engine.
- For all physically replicated tables, Brad must be able to replicate writes quickly enough to meet the VDBEs' freshness constraints.

Brad applies a search strategy driven by learned performance models to find an optimized design [9]. These chosen designs are not static. If the workload changes and triggers a user-set condition (e.g., a VDBE's p90 query latency SLO is violated for 100 consecutive queries) Brad will redo this optimization to select a new design [9].

2.5 Physical Design Options for QuickFlix

Coming back to our example, Figures 1 Ⓑ, Ⓒ, and Ⓓ show three possible physical designs for QuickFlix's VDBEs. We could map all three VDBEs onto a single Aurora engine (Figure 1 Ⓑ). This design is cost-efficient and would provide acceptable performance at low workload scales [9]. Another option is to use two engines: Aurora for transactions on `showings` and `tickets` and Redshift for analytics on `view_hist` (Figure 1 Ⓒ). This design is more expensive but provides better performance at higher workload scales [9]. A third option is to use Athena, a serverless engine, for VDBE 2 (Figure 1 Ⓓ). This design makes sense if QuickFlix's transformations require much more compute resources than their analytics. In that case, offloading the transformations to Athena would be less costly than using a larger Redshift cluster for both VDBEs 2 and 3. An automated planner would pick the best design based on the workload and QuickFlix's cost constraints. We note that even if QuickFlix chooses to manually design their infrastructure, using VDBEs gives them the flexibility to change their physical design later on without having to modify their application code.

3 Demonstration Scenarios

Using three scenarios based on real-world data from the IMDb dataset [6], our demo will showcase the VDBE abstraction and Brad's automated planner. As shown in Figure 2, Brad's dashboard visualizes its VDBEs Ⓐ, their physical realization Ⓑ, and real-time performance metrics observed on each VDBE Ⓒ. Users can have Brad predict the impact of workload changes on the physical

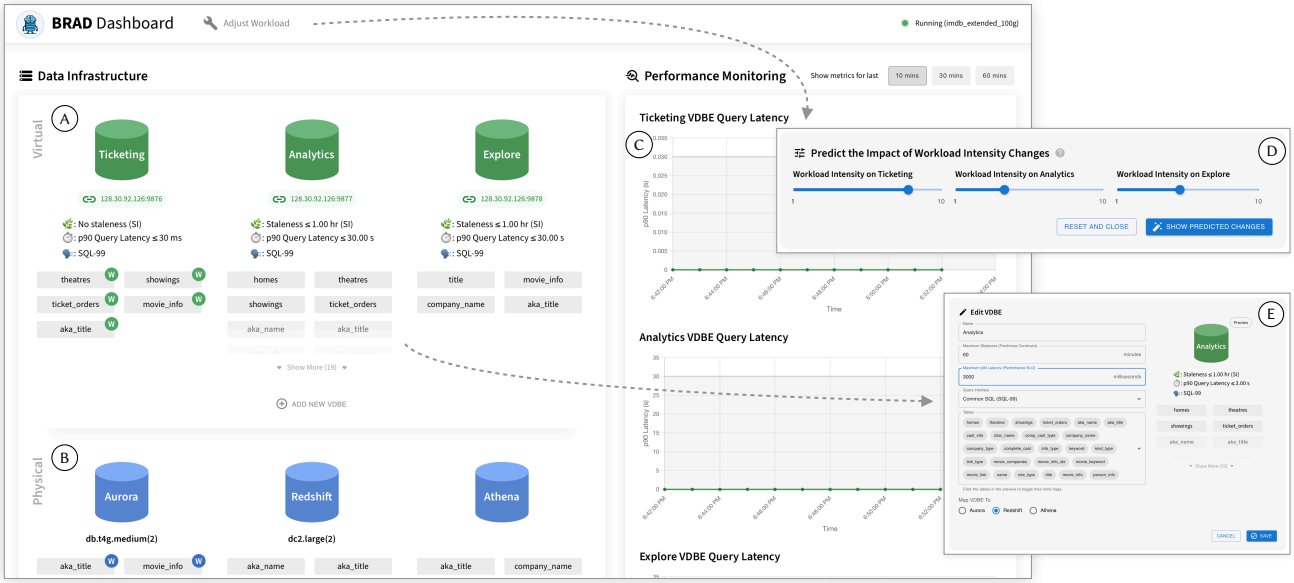


Figure 2: A screenshot of Brad's interactive dashboard.

infrastructure (D) and manage the VDBEs (E). A video showing our demonstration is available here [8].

3.1 Automated Planner: Workload Changes

In this scenario, we will show Brad's automated planner in action under workload changes. Recall that given a set of VDBEs and a workload, Brad's planner automatically selects an optimized physical design and can change it if the workload shifts. Attendees will drag sliders on the dashboard to convey a hypothetical change to the workload. To keep our demo interactive, Brad will then show (but not deploy) the physical design that it selects for the VDBEs under the changed workload.

Concretely, we will create two VDBEs (T and A) with the same tables. VDBE T will support transactions with zero data staleness and have a query latency SLO of $\leq$ 30 ms. VDBE A will have a maximum staleness of 1 minute and a query latency SLO of $\leq$ 30 s. This setup is common in HTAP workloads. Attendees will drag sliders to modify the workload intensity on each VDBE. At a low workload intensity, Brad will map both VDBEs onto a single Aurora engine as it can handle the full workload. At a high workload intensity, Brad will start up Redshift and move VDBE A over to it.

3.2 Modifying the Virtual Infrastructure

Although Brad automatically selects the physical infrastructure, attendees will be able to manually modify the virtual infrastructure and its physical realization to get a feel for the operational simplicity of VDBEs. On the dashboard, they will click a button to add a new VDBE and will get to customize its properties (e.g., the tables on the VDBE) and map it to a physical engine. The dashboard will show the VDBE's endpoint and attendees will connect to the VDBE using a SQL client and will be able to issue queries to it. While the client is connected, the attendee can also modify the VDBE's engine mapping and continue to issue queries. By switching among different engines, the attendee will get a feel for the different performance characteristics of each engine without having to disconnect their SQL client. They will also be able to remove VDBEs and modify the other existing VDBEs in the infrastructure.

3.3 Supporting External Tools: Data Pipelines

External tools (e.g., AWS Glue [1], dbt [4], etc.) work with VDBEs like any other database engine. Brad provides ODBC/JDBC drivers that data tools can use to read from and write to specific VDBEs. To show how this integration works, attendees will get to build a data pipeline across VDBEs using AWS Glue. We will start with just VDBEs 1 and 3 as shown in Figure 1. We will then create a Glue job that extracts `tickets` and `showings` from VDBE 1, processes them, and then loads the result into VDBE 3 as `view_hist`. Attendees will be able to tweak the Glue job and launch it. Glue accesses the VDBEs by connecting to their endpoints using Brad's JDBC driver. After the job completes, we will query VDBE 3 to show that the transformed data is visible and reflects the writes made on VDBE 1.

References

[1] Amazon Web Services. 2025. *What is AWS Glue?* https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html.
[2] Amazon Web Services. 2025. *zero-ETL.* https://aws.amazon.com/what-is/zero-etl/.
[3] Pgbouncer Authors. 2023. Pgbouncer. https://www.pgbouncer.org/.
[4] dbt Labs. 2025. dbt. https://www.getdbt.com/.
[5] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X. Yu. 2023. Check Out the Big Brain on BRAD: Simplifying Cloud Data Processing with Learned Automated Data Meshes. *VLDB '23* (2023).
[6] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good are Query Optimizers, Really? *VLDB '15* (2015).
[7] Amadou Latyr Ngom and Tim Kraska. 2024. Mallet: SQL Dialect Translation with LLM Rule Generation. In *aiDM '24* (Santiago, AA, Chile).
[8] Geoffrey X. Yu et al. 2025. Virtualizing Cloud Data Infrastructures with BRAD (Demo Video). https://youtu.be/6IaspuIYgRg.
[9] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *VLDB '24* (2024).