Research Statement

Matthew Perron

Analytical database systems allow users to ask complex questions over large volumes (TB to PB) of tabular data. These system abstract away difficult decisions such as data storage, partitioning, and query execution behind a simple declarative language. The database system handles the complexity of where to store data and how to process data to answer user queries. Visualization tools like Tableau build atop analytical databases, removing the need to know SQL to leverage analytical databases. These tools are used daily by nearly all enterprises to gain insight into their businesses. The current valuation of analytical database companies like Snowflake and Databricks, both valued around $50 billion, reflect their current and perceived future importance in organizations data infrastructure.

Historically, analytical databases were deployed in users' own data centers. This made increasing or decreasing the computational resources of the system slow and cumbersome, as machines had to be physically added to the system. Thus these systems were provisioned to handle the peak expected query loads on these systems. Data also had to be re-balanced to new machines. As users moved to the cloud, adding and removing resources became much easier, as users could now request new new hardware with the click of a button. Furthermore, "cloud-native" analytical databases decoupled storage and compute. Instead of storing data on local disks, these systems instead use inexpensive cloud object storage services like Amazon S3. These systems cache user data on local storage media and in memory for performance rather than persistence. This has made systems significantly more **elastic** as whole clusters of machines can be added or removed in seconds. But with the exception of storage disaggregation and an increase in elasticity, the architecture and deployment of these systems remains much as it was when they were deployed on-premises in customer data centers. For instance, individual queries can only use the amount of hardware provisioned for a specific database instance.

Despite improvements made after moving to the cloud there remains a mismatch between the needs of analytical database users and the way these systems are constructed and used. Users have diverse workloads that can now query any data stored on cloud storage at any time. Workloads have frequent and difficult to predict spikes in resource demand while the systems are still provisioned in a static or slowly adapting way. These spikes may come from users who have business critical queries that need vast amounts of resources to answer promptly. Because the source of load is an human users, there is no way to predict precisely when this increase in load will arrive.

Handling this mismatch is currently left to users who must decide how many and what size of database instances to provision and how to direct queries in their workload to different database instances. Choosing incorrectly can result in an order of magnitude worse query latency than optimal if the system is overloaded by a workload spike. It can also result in an order of magnitude higher cost if users provision for peak load. Dynamically changing workloads make choosing an efficient provisioning of resources nearly impossible. Many aspects of cloud analytical databases leave complex decisions like these to users.

Complexity of this kind should be completely managed by systems. Analytical databases should tune themselves to meet user demands rather than leaving users to deal with this complexity. To this end, I have built systems and designed methods to match provisioning to demand, while being sensitive to the variable cost inherent to cloud environments. **My research aims to drastically improve the experience of cloud analytical database users by leveraging the unique properties of cloud environments to meet user needs.**

1 Previous Research Experience

1.1 Starling

I built Starling [1] to address the mismatch of workload resource requirements and provisioning of these resources. The key idea of this project is to address the match resource demand of a workload and supply by leveraging elastic cloud function services like AWS Lambda. Cloud function services provision compute resources to execute a user's custom code with latency in the 10s to 100s of milliseconds, scaling up to thousands of cores and down to zero again near instantaneously. Critically, these services are billed at fine granularity, allowing users to pay only for what is used. However, these services come with some limitations, including limited run-time, small granularity, and restrictions on communication. Furthermore, on a per CPU-time basis, using cloud function services can be nearly an order of magnitude more expensive than an equivalent provisioned virtual machine.

Like some other analytical systems, Starling executes queries by transforming them into a directed acyclic graph of stages, each with one or more tasks. Between each stage data must be exchanged through a shuffle operation. Because of the lack of direct communication in cloud function services, Starling shuffles data through a cloud storage service like Amazon S3, increasing cost the. Starling schedules each task on a cloud function as soon as its inputs are available. The overall reduction in cost comes despite the increase in per-second compute costs and shuffling costs. They are a result of compute resources sitting idle between queries in a fixed-provisioning. But there is a further advantage of Starling, unlike a fixed size cluster, which is common practice in analytical databases, Starling acquires variable amounts of compute as needed by each query or stage. Thus it can behave as a large cluster or a small one based on the needs of the query. This removes the need for the user to choose a cluster size. This is particularly useful when a workload requires vastly different amounts of compute. Starling acquires compute when needed and releases it immediately when tasks complete. Starling was nearly the same performance at significantly lower cost than alternative commercial systems on infrequent workloads.

I implemented Starling atop Amazon infrastructure and compared against the market prices of their services, but the lessons of this project are wider. Specifically, Starling demonstrates the value of analytical database systems being vastly more elastic than current approaches. Indeed the results of Starling show that gain in near-instant elasticity results in significantly lower cost for infrequent workloads *in spite of* the restrictions of current cloud function services. The gain in flexibility is large enough to outweigh the costs. Rather than users needing to **predict** future load which often leads to a mismatch in demand and supply of resources, Starling **reacts** to changes in demand. It is able to do this because the latency of acquiring new resources is so low.

This project has been influential in the research community. Since publication in 2020, This work has been cited nearly 100 times, and is often compared against when building data systems atop cloud function services. It has also attracted some attention from industry where I was invited to give talks to both Databricks and IBM.

1.2 Cackle

Starling showed the value of rapid burst-out elasticity for analytical database systems, but it also showed that a system that relies on more expensive compute and shuffling resources will, unsurprisingly, be more expensive when workloads are static or slowly changing. The question I address in Cackle [2] is how to make use of burst out elasticity for a much larger range of workloads. As I found in the Starling project, current commercial "cloud-native" solutions to tackle elasticity move relatively slowly to address changes in resource demand. This results in queries queuing up while new resources are provisioned, and significantly reduced end-to-end query latency. If these workloads are mission critical, users must ensure that these systems are consistently over-provisioned to avoid this queuing. In Cackle I propose a hybrid strategy that uses both provisioned virtual machines, as well as burst-out scalable compute, as found in cloud function services.

In the cloud performance can usually be gained by adding hardware, and thus Cackle focuses on reducing the cost of executing workloads. Specifically, Cackle decides how many virtual machines to provision to a workload. Any demand exceeding this workload are served by an "elastic pool" of compute like cloud function services. While provisioned machines are less expensive than cloud functions, they take more time to start up, and they often come with minimum billing requirements of a minute or more. Thus bursty workloads are best served by elastic but more expensive cloud functions, while consistent workloads are better served by less expensive provisioned virtual machines.

The main insight of Cackle is that the best strategy for allocating virtual machines depends on three factors: (1) Workload Properties, including spikiness, growth over time, and cyclicality (2) The Cost differential between virtual machines and the elastic pool and (3) Environmental properties including VM startup time, minimum billing time, etc. Cackle employs a meta-strategy that selects the best among a family of simple allocation strategies with a multiplicative weights algorithm using historical data. In an analytical model I demonstrated that this meta-strategy achieves close to the performance of an oracle with full knowledge of future workloads in a range of realistic scenarios.

I implemented the ideas of Cackle atop AWS Spot instances, AWS Lambda, and designed and built a simple intermediate shuffling services that significantly reduced the cost of shuffling intermediate data compared to Starling. Cackle achieves better cost and performance stability across a range of workloads than commercially available alternatives, despite the infrastructural baggage of using cloud functions, and the need for an external shuffling layer. These restrictions could potentially be eliminated if the ideas were adopted by a cloud analytical database with it's own multi-tenant elastic pool.

Cackle is a step toward simplifying one of the most important and difficult to tune parameters in cloud data analytics: provisioning.

A paper on the Cackle project has been accepted at SIGMOD 2024.

2 Research Goals

While I have worked towards building systems that drastically simplify the user experience of cloud analytical data systems, A number of simplifications could still be made to the user experience of these systems.

2.1 Cache Aware Elasticity

One of the main innovations of "cloud-native" analytical systems is storage disaggregation. Data primarily resides in cloud object storage and is cached in memory and local storage of compute clusters. This improves query latency when the workload has significant reuse of data. Using cloud object storage simplifies adding and removing hardware to the system. However, users must still decide the size of clusters, and where to direct queries to maximize the cache hits on these localized caches. This is currently a manual process that users lack detailed statistics to solve. Making sub-optimal decisions can lead to low utilization of caches and thus poor query performance. Furthermore as workloads change over time these decisions need to periodically reevaluated. Instead, I believe analytical databases should handle this complexity for users by observing patterns of data accesses, and handle scaling of hardware and partitioning of queries to maximize cache utilization and thus minimize the amount of hardware required, saving on cost as well as improving query latency. Instead of making these complex decisions, I think users should have a single query interface, with the system handling the complexity of maximizing cache hits.

2.2 Cloud Query Optimization and Scheduling

Historically the goal is to make the best use of the fixed set of hardware available to reduce query latency. The cloud opens elasticity as a new dimension to optimize. With an essentially unlimited set of resources available in the cloud, including through cloud functions, query optimizers can choose to allocate more or less hardware at any time. In the Starling and Cackle projects, queries were hand-tuned to provide a reasonable cost and performance trade-off. But in some cases users may want to pay more to achieve lower latency and are willing to pay more to achieve this. In other cases, users are willing to suffer lower latency to save on cost. Current approaches can give priority scheduling to some queries scheduling to such queries, but cannot acquire new hardware or adjust the query plan to handle these demands. Ideally users should have a way to express these priorities and the system should adjust accordingly, isolating queries with low latency requirements.

2.3 Dynamic Query Reoptimizaton

Query Optimizers often produce plans that are orders of magnitude slower than optimal. While there has been a research trend in improving optimizers, an alternative approach is to reoptimize plans during execution using statistics from execution improve these results. Early in my PhD I investigated the value of query optimizations against recent optimization approaches [3]. In traditional approaches, query execution is paused while a query is re-optimized as to not waste resources. In cloud environments, rather than pausing execution we can instead try alternative plans on additional hardware hardware. If no better plan is found the original plan can run to completion, without interruption or interference from this exploration. For performance sensitive queries, how many alternative plans should be considered and under what conditions remains an under-explored research question. This project would attempt to find reasonable cost and performance trad-offs that depend on market prices for acquiring new hardware as well as how much a user is willing to pay to achieve lower cost. In cloud environments, the trade-off may be between allocating more resources to a sub-optimal plan or by exploring the space of plans to find more efficient ways to use current resources. Improving this would improve the user experience of analytical data systems by giving users options to spend more to reduce latency on a per-query basis which is absent from current systems.

2.4 Spill to Elastic Resources

When query execution requires more memory than what is allocated, query engines ”spill“ intermediate state to disk, this allows systems to continue processing even when the size of intermediate state exceeds that of available memory. However this comes with a significant performance penalty as data must be constantly moved back and forth between memory and storage, thus also adding I/O pressure. Because elastic pools, like cloud functions, allow users to acquire new resources so quickly, systems might instead choose to spill intermediate state out into the memory of newly acquired resources, possibly avoiding the overhead of spilling to disk. How exactly to build such a mechanism and what the current cost and performance tradeoffs are are currently unknown. Improving this

2.5 Better Utilization Through Market Pricing

Cloud environments are well known to be vastly under-utilized. I believe that this under-utilization results from users having no direct way to specify their performance and cost goals. Instead users choose things that are related to this goal indirectly. Data systems provide a case study. Caching table data can result in vastly improved query latency, but holding onto this cache when users do not need this performance is a waste of resources. Currently users do not have convenient ways of directly expressing these preferences for performance or cost. It is difficult to determine preference based on usage behavior alone. This inability to specify needs in a direct way results in users holding onto resources even when they are not needed, as there is no convenient alternative. This is made more complicated as the underlying prices for these resources change. A system should behave differently when resources are expensive vs inexpensive based on user preference. For instance, with inexpensive resources a user may want to cache in memory their entire working set, but when resources are more scarce, the system may fall back onto caching on local disk, or always reading from cloud storage. This under-utilization drives the cost of the cloud up for everyone as demand is artificially inflated. I want to build data systems that take into account user preferences in a first class way so that resources are acquired and held only when needed to meet a performance target that users are willing to pay to meet.

References

[1] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery.

[2] Matthew Perron, Michael Cafarella, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Cackle: Analytical workload cost and performance stability with elastic pools (to appear). *Proc. ACM Manag. Data*, 1(1), December 2024.

[3] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. How i learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1758–1761. IEEE, 2019.