The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward

Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, Amrish Kumar*
Microsoft
firstname.lastname@microsoft.com

ABSTRACT

The twenty-first century has been dominated by the need for large scale data processing, marking the birth of big data platforms such as Cosmos. This paper describes the evolution of the exabyte-scale Cosmos big data platform at Microsoft; our journey right from scale and reliability all the way to efficiency and usability, and our next steps towards improving security, compliance, and support for heterogeneous analytics scenarios. We discuss how the evolution of Cosmos parallels the evolution of the big data field, and how the changes in the Cosmos workloads over time parallel the changing requirements of users across industry.

PVLDB Reference Format:
Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, Amrish Kumar. The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. PVLDB, 14(12): 3148 - 3161, 2021.
doi:10.14778/3476311.3476390

1 INTRODUCTION

The world is one big data problem.
— Andrew McAfee

The last decade was characterized by a *data deluge* [78] in large enterprises: from web, to social media, to retail, to finance, to cloud services, and increasingly even governments, there was an emergence of massive amounts of data with the potential to transform these businesses and governments by delivering deeper insights and driving data-driven decisions. Unfortunately, prior tools for data processing were found to not work for this scale and complexity, leading to the development of several so-called big data systems. At Microsoft, the big data system development started with large-scale data extraction, processing, and analytics in Bing, resulting in a compute and storage ecosystem called *Cosmos*. Over the years, Cosmos grew into a mammoth data processing platform to serve the fast-evolving needs for big data analytics across almost all business units at Microsoft. Figure 1 illustrates this growth in terms of the number of servers, the total logical data before replication or compression, and the total number of batch SCOPE jobs. Indeed, we can see a phenomenal growth of 12x, 188x, and 108x on these three metrics over the course of the last ten years.

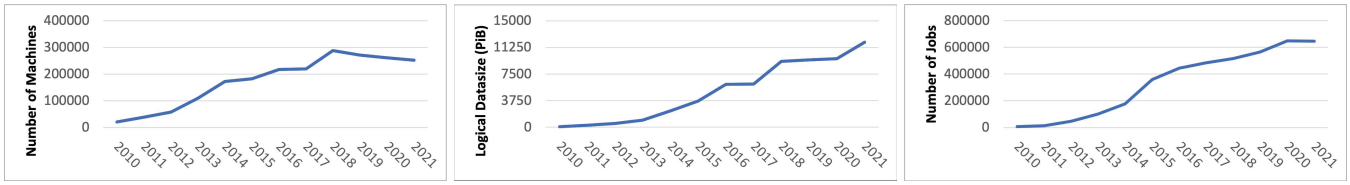
In this paper, we look at the tremendous progress in storing and processing data at scale that has been made at Microsoft. We trace all the way back to early efforts starting in the early 2000s and describe how they lead to the Cosmos data processing platform that has been the analytics backbone of the company for the last decade. In particular, we highlight the technical journey that was driven by constantly evolving user needs, starting from store reliability, running compute over the stored data, developing a declarative interface, the origins of the SCOPE language, details about SCOPE input processing, including appends and metadata, job characterization and virtual clusters for concurrency, network challenges seen and the corresponding optimizations, the origins of the SCOPE query optimizer, and finally, the transactional support in Cosmos.

After the initial years of development, post 2010, the core Cosmos architecture has remained relatively stable and has been serving a broad spectrum of analytics needs across the whole of Microsoft, including products such as Bing, Office, Windows, Xbox, and others. We describe several aspects of this core architecture, including the design for hyper-scale processing; compiler re-architecture to align with the C# specification and semantics; supporting heterogeneous workloads consisting of batch, streaming, machine learning, and interactive analysis; high machine utilization exceeding 70% in most cases and 90%+ in many cases as well; and a comprehensive developer experience including tools for visualization, debugging, replay, etc., all integrated within the Visual Studio developer environment. In recent times, Cosmos has further witnessed several technological advances and has been extended to support several modern needs, including the need for better efficiency and lower costs, adhering to newer compliance requirements such as GDPR, embracing open-source technologies both in the platform and the query processing layer, and opening to external customers to serve similar analytics needs outside of Microsoft. We describe these recent extensions and discuss our experiences from them.

Looking forward, we expect Cosmos to remain a hotbed of innovation with numerous current and future directions to address tomorrow's analytical needs. Examples include continuing to address the challenges around security and compliance; providing an integrating ecosystem within Cosmos with more flexible resource allocation and tailored users experiences; better integration with the rest of the Azure ecosystem to support newer end-to-end scenarios; richer analytical models such as sequential and temporal models for time series, graph models for connected data, and matrix

*Equal Contribution from first three authors

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476390



(a) Number of servers in Cosmos clusters. **(b) Data size before compression/replication.** **(c) Number of batch SCOPE jobs run per day.**

Figure 1: The growth of Cosmos infrastructure and workload in the last decade.

models for linear algebra operations; supporting advanced analytical workloads that are a blend of data science, machine learning, and traditional SQL processing; providing a Python language head on top SCOPE; improving developer experience with better unification, interactivity, and recommendations; optimizing the new class of small jobs that are now a significant portion of Cosmos workloads; applying workload optimization to reduce the total cost of ownership (TCO) for customers; and finally, leveraging the recent advances in ML-for-systems for tuning difficult parameters and reducing the cost of goods sold (COGS).

Despite the long development history, interestingly, Cosmos remains very relevant to the modern big data trends in industry. These include ideas such as “LakeHouse" [6] for better integration between ETL and data warehousing, “Lake Engine" [22] for advanced query processing on the data lake itself, “Data Cloud" [75] for democratizing the scale and flexibility of data processing, and “Systems-for-ML" for bringing decades of data processing technologies to the machine learning domain. Therefore, we put Cosmos in context with these newer industry trends.

In summary, efficiently storing and analyzing big data is a problem most large enterprises have faced for the last two decades. Cosmos is the Microsoft solution for managing big data, but many other companies have built their own internal systems. Most notably, Google built Map Reduce [21] and the Google File System [25] before moving to Colossus and Google Dataflow [43]. Other companies have built their internal solutions on top of open-source Hadoop [30], such as Facebook with its Hive-based solution [77], as well as LinkedIn [76] and Twitter [44] [46]. Many of these internal solutions have been later offered as big data services to external customers, such as Microsoft Azure Data Lake [50], Google Dataflow [27], and Amazon Athena [3]. Our goal in this paper is to share the rich history of Cosmos, describe how the system and workloads have evolved over the years, reflect on the various design decisions, describe the next set of transformations we see in the big data space, and contrast traditional big data systems like Cosmos with the latest data processing trends in industry. To the best of our knowledge, this is the first paper discussing both historical evolution and modern relevance of a production big data system.

The rest of the paper is organized as follows. Section 2 traces back the origins of Cosmos between 2002-2010, Section 3 describes the core architecture between 2011-2020, Section 4 describes challenges and opportunities we see in 2021 and going forward, finally Section 5 puts Cosmos in context with modern industry trends.

2 THE ORIGINS: 2002-2010

> *Those who don’t learn history are doomed to repeat it.* — George Santayana

In this section, we delve into the early origins of Cosmos and describe the various design choices made along the way, in response to both the customer needs and the operational challenges.

2.1 Storage Efforts

The origins of Cosmos can be traced back to early 2002, with the need for a reliable and efficient data store. Early efforts included the XPRESS library for fast compression using LZ77 [89] and optionally adding a Hoffman or other encoding pass, the Lock Free (LF) library for various components in the store, such as config manager, object pool, allocator, releaser, etc., and the Replicated State Library (RSL) [55] as a Paxos [10] implementation to support dynamic replica set reconfiguration, including dynamic sizing and cluster healing. These ideas evolved into Cosmos store in 2004 and the original goal was to achieve a cost efficient, high available, and high reliable storage solution for user mailboxes in Hotmail.

Cosmos was later incubated in Bing in 2005, then called Live Search, where the initial code for several components was added, including *extent node* (EN) to manage data on a single machine and communicate with other machines (built on top of NTFS), *Cosmos storage manager* (CSM) to keep metadata about streams and extents in a Paxos-protected ring with seven replicas that keep all metadata in RAM, *Clientlib* interface to call EN and CSM, and *Cosmos Web Service* (CWS), to browse directories and streams. The original design had many JBOD (Just a Bunch of Disks) ENs, 100MB extents (soon 250MB) compressed and stored in triplicate in different failure domains, multiple CSM volumes all using the same ENs, a Clientlib, and the ability to run a distributed computation. There was no distributed computation framework though, and users had to write it all by themselves. A distributed CSM was supposed to hide volumes of distributed storage from users but was not in the initial implementation. Initial customers were Books, Boeing Blueprints, and Search Logs. In 2007, the CSM, EN, and Clientlib components of Cosmos were forked into an internal codebase called Red Dog [23], which later became Azure Storage [12, 17].

2.2 Compute Efforts

In 2006, a Nebula Algebra was developed to run computations on Cosmos store. The algebra consisted of stages, which had code to execute, along with inputs and outputs. For example, a join would have two inputs and one output. A job hooked together the inputs and outputs of stages to read/write from Cosmos store. Each stage

could have a variable number of *vertices*, which ran the stage's code single-threaded on a particular machine. There was fan-in for each input and fan-out for each output, mapping the M vertices from one stage to the N vertices of the next stage. Intermediate outputs were written as distinctively named files on the machine where a vertex ran. Users could write an algebra file describing the stages, and the system could run the distributed job described by the algebra file. Later, Microsoft developed Dryad [34], which was a more robust method of managing execution trees that could retry subtrees on failures, and so the Cosmos team switched to using Dryad to execute the Nebula algebra. However, authoring the algebra file was still not easy.

2.3 Declarative Language

The Search team developed P*SQL, meaning Perl-SQL, which generated a Nebula Algebra file from a SQL-like script with embedded Perl fragments. The compiler grepped the script for "SELECT", "FROM", etc., and everything in between was assumed to be Perl. The Search team used it to mine its logs to find most frequent queries and responses people clicked on. The earnings helped Cosmos pay for itself in a month. The purpose of Cosmos now changed to running large, distributed jobs instead of just storing data reliably. However, P*SQL was clunky. It just searched for keywords and assumed the things in between were legal Perl. `PSQLv2` smoothed user experience, but users still struggled to work with the complexities of Perl language. This led to the birth of `FSQL`, which had `F` expressions and supported nested tables.

In 2007, Microsoft invented `DiscoSQL`[66], which was like P*SQL but used C# snippets instead of Perl snippets, and it had a GUI that allowed dragging-and-dropping stages together into an algebra instead of requiring a script. By default, each statement used the previous statement's output as its input. Alternatively, statements could be assigned to variable names, and those variables could be used as inputs to multiple later statements. Input streams were binary, but "extractors" interpreted them as a list of records of known types; "processors", "reducers", and "combiners" took lists of records and produced lists of records. Unlike MapReduce [21], combiners took two input lists of records and produced one output list of records. "Outputters" took a list of records and translated them into binary streams. Schemas were embedded in the extractor code rather than the inputs or the metadata store. Cosmos eventually standardized on `DiscoSQL`, which was later renamed `SCOPE` and has remained the primary language of Cosmos since then.

2.4 SCOPE Beginnings

Structured Computations Optimized for Parallel Execution, or SCOPE, was initially a *grep-for-select* language just like P*SQL, but with C# fragments instead of Perl fragments. It had both a GUI and a script language. However, development in 2007 and 2008 centered on just the script language (not the GUI), giving it a well-defined syntax. The design philosophy of SCOPE was SQL clauses with C# expressions. Although SQL is not case sensitive, SCOPE was made case sensitive with all uppercase keywords, to not conflict with C# ("AS" is in SQL while "as" is in C# and SCOPE had to support both). `DiscoSQL` already supported user-defined extractors, processors, reducers, combiners, and outputters. SELECT was added to SCOPE as syntactic sugar. All types of joins were implemented as internal combiners. Aggregate functions were implemented on top of reducers. DefaultTextExtractor and DefaultTextOutputter made reading and writing tab-separated and comma-separated Cosmos streams easy. Users loved the C# "yield" statement, so that was used for user-defined operators to report rows. If an expression had an error, raising an error would fail the job, which is often too expensive. Nullable types were added to SCOPE, allowing users to report `null` for an erroneous expression, rather than fail the entire job.

The SCOPE language was designed to have a tokenization pass (lexer) and a parsing pass (grammar). The grammar took a list of tokens as input, never had to retokenize (tokenizing is expensive). Tokenization depends on whitespace, but grammar does not. Tokens remembered the following whitespace (comments count as whitespace), and their position in the script. The token positions, and following whitespace, could be used by error messages to point at where errors were and what the surrounding text was. Compilation figured out how statements were hooked together and split statements into operations. Optimization rearranged the operations for improved execution speed, e.g., filtering was done as early as possible. Internal operations, and even user-defined operations, could annotate whether various transformations (like pushing filters earlier) were legal. C# code was generated to implement each operation. Operations were grouped into stages that could execute single-threaded on a single machine. The output of compilation was C# code and a Nebula algebra file that described how stages are hooked together, and the number of vertices for each stage.

Later, SCOPE started having a dedicated editor that would parse the SCOPE script and give intellisense suggestions. An experimental feature annotated the compiled script with line numbers from the text so that a debugger in Visual Studio could report what line number in the UDO in a script was causing a problem. SCOPE can also run over local files on the developer machine.

2.5 Extractors, Appends, & Structured Streams

SCOPE processes inputs using "extractors" that break a byte stream into records with a well-defined schema. The schema contains C#-nullable types. SCOPE extractors read whole *extents*, i.e., the chunks into which a Cosmos stream is subdivided, and so records could not span extent boundaries. There was an early bug in the Cosmos data copying tool that would break the alignment of records along extent boundaries. This failed customer jobs and inspired them to design customer extractor UDOs. Users were further motivated to share these custom UDOs with each other, and thus UDO sharing quickly became an essential workflow in the Cosmos ecosystem.

Earlier, users would place the libraries into the Cosmos store and then share the location, but this made it hard to keep the libraries public while keeping the data private. In recent years, with the addition of the metadata services (see section 3.9), customers now can share the code objects using a common pattern. Even the Cosmos team can provide commonly used user-defined operators as built-in assemblies. Finally, inputs can either be extracted (each extractor vertex gets a subset of the stream's extents), or they can be resourced (copied as files to every vertex of the job). Copying a resource to so many vertices could overwhelm Cosmos, so a P2P file

copy service called Databus was used to fan out resources rather than having each vertex copy them directly.

Cosmos supports both fixed-offset and unknown-offset appends. In fixed-offset appends, the offset to append at must be known, or the append fails. Fixed-offset appends will succeed if the data at that offset is already there and matches the data being appended. Fixed-offset appends can guarantee no duplicates, but need a single writer. On the other hand, unknown-offset append cannot avoid doing duplicate appends, but they can be done in parallel. SCOPE outputters use fixed-offset appends, each outputter writes its own extents, and then extents are concatenated together into a final output stream. This allows SCOPE to produce large, sorted outputs in parallel, with no duplicates. Appends were limited to 4MB, but in 2010 large appends that could be up to 250MB were implemented. They did repeat 4MB appends by offset and failed all of them if any of them failed. Large appends fail more than normal appends, and they are particularly sensitive to flaky networks.

Finally, SCOPE implemented "structured streams", which tracked their own metadata in a stream's final extent. Such streams could use schema-aware compression. They were automatically affinitized into partitions, and extractors read whole partitions. Records could span extents within a partition because the extractor would read the extents within a partition sequentially.

2.6 Jobs & Virtual Clusters

SCOPE jobs could be classified as cooking jobs or ad-hoc queries. Cooking jobs are massive, with the largest at the time being the daily merging of all clicks and queries from Search. The Search team put a lot of effort into extractors to interpret the heterogeneous logs, join click logs with the query logs, and produce clean non-duplicated outputs. The output of cooking jobs was about as big as the input, but sorted, cleaned up, and without duplicates. On the other hand, ad-hoc jobs typically spent 90% of their CPU in massively parallel extractors. The extract vertices discard most of the data, aggregate most of the rest, and then later vertices spend a long time churning that remaining data to produce a final answer as output. Most of the CPU went into parsing tab-separated-value files and sorting the results. An example ad-hoc job is to query all search queries and the links clicked on for the past three months.

Customers also found interesting ways to exploit SCOPE jobs. For example, some jobs included replacements for the cosmos internal executables that executed them as resources. Others attempted to read the internet, or the job vertices used more space, more CPU, or wrote to disk directly. Still others included applications for machine learning and model training. Given the unusual ways customers were using Cosmos, it was challenging to test the impact of code changes on production jobs. The solution was to implement "playback" which allowed to recompile and re-run the jobs in production environment [82]. We store the metadata for each SCOPE job in a job repository so that we can verify backward compatibility and detect performance regressions before deploying new code.

Initially, Cosmos clusters could only execute one job at a time. This was fine for baking search query logs, but not acceptable for numerous other scenarios. Therefore, *virtual clusters* (VCs) were introduced to allow multiple concurrent jobs (still one job per VC). VCs also functioned to hide volumes, and a way to hide one user's data from another. VCs were given a quota of a fixed number of machines they could use to execute jobs. A quota of 15 machines wouldn't always be the same 15 machines, but only 15 machines at a time could be executing a VC's job. The VC Web Server (VCWS) provided a VC-oriented interface to Cosmos.

2.7 Network Traffic & Optimizations

Early Cosmos cluster networks were woefully underpowered and minimizing network utilization was a major focus. For example, to make SCOPE practical, extractor input extents were sorted by the machines they were on, extents were grouped into vertices, and vertices were placed so that extractors read almost all the time from their local machine. This avoided network traffic for extract, which is where 90% of the data is read and discarded. Likewise, mirroring was added to continuously copy data from one cluster/volume to another cluster/volume. This was needed to migrate from old clusters to new ones and rebalance volumes within a cluster. Mirroring used the same underlying mechanisms as cross-volume and cross-cluster concatenate. Another network optimization was to map the first extent to three random racks, but map later extents with the same affinity GUID to the same racks. A vertex reading several affinitized extents could be placed in one of those racks and would not need cross-rack network to read any of its data. A vertex joining two affinitized streams could join them by their affinitized keys without reading cross-rack data. An "Affinity Service" kept affinity groups in sync across clusters.

Cosmos store also experimented with 2x2 replication (two copies in one cluster, two in another), but this proved insufficient for availability during outages. Plain two-replica extents were also supported, but they lost a trickle of data constantly. Replicas are lost when servers die, or they need to be reimaged for repairs. Each extent replica is kept in a separate failure domain, so correlated hardware failures, such as power failures, do not cause the loss of every extent replica. When one extent replica is lost, the CSM creates a new replica to replace it, so data loss occurs when all replicas are lost within the time window it takes for the CSM to do this replication. In practice, we observe that these data loss scenarios occur with two replicas, but not with three replicas.

Other efforts for making the Cosmos store more efficient include Instalytics [74] for simultaneously partitioning on four different dimensions with the same storage cost as three replicas, and improving tail latency for batch workloads in distributed file system [58].

2.8 Query Optimizer

In 2008, the cost-based query optimizer from T-SQL was ported to optimize the plans for SCOPE queries. SCOPE could query all streams matching a certain directory and stream name pattern and treat them as if they were just one big stream. This was vital to cooking jobs which had to read hundreds of streams written in parallel for ingested data. It was also vital to ad-hoc jobs reading weeks' or months' worth of cooked streams in date-structured directories. Internal joins were limited to 1000x1000 results per matching key. If both sides had over 1000 rows with the same key value the job failed with an error, but it was fine if one side had 999 rows and the other had 1,000,000. This rule was to prevent small inputs from causing exceedingly expensive jobs. Combines without

a join key (cross product) were also disallowed for the same reason. Over time, improvements in the SCOPE optimizer led to numerous improvements in job execution. SCOPE switched to a grammar-driven syntax and fully parsed C# expressions. Error messages improved, user load grew orders of magnitude, some scripts were thousands of lines long, and many scripts were machine generated.

2.9 Transactional Support

In 2009, a service called Baja was developed on top of the Cosmos store file system to support ACID transactions. The Baja project started by forking the partition layer of Azure Storage [12], a service that itself had started as a fork of the Cosmos store. Baja added support for distributed transactions and a distributed event processor. The initial use case for Baja was to reduce index update times for Bing from minutes to seconds. This was particularly important for updating the index for Bing News. The Baja solution was highly scalable and cheap, so Bing was able to use it for many indexes beyond News. One of the taglines was "the entire web in one table", where the key is the URL. Baja supported tables of unlimited size by implementing an LSM tree on top of Cosmos Structured Streams. Baja used a custom format for the log but checkpointed the log to standard Cosmos Structured Streams. A streaming SCOPE job was then used to do compaction of separate Structured Streams into a new Structured Stream. Baja was later used to support other scenarios, outside of Bing indexing, across Microsoft. One major scenario was Windows telemetry. After a successful run of ten years, the Baja project evolved into a solution that runs on Bing servers rather than Cosmos servers.

3 THE CORE DESIGN: 2011-2020

> *You can't build a great building on a weak foundation.*
> — Gordon B. Hinckley

In this section, we describe the core Cosmos architecture that has remained relatively stable over the last decade. We discuss the scalable compute and storage design, the modern compiler architecture in SCOPE, support for a highly heterogenous workload, the high machine utilization seen in Cosmos, and a relentless focus on the end-to-end developer experience. We further describe recent advances, including efforts to improve operational efficiency, making Cosmos GDPR compliant, bringing big data processing to external customers via Azure Data Lake, designing a unified language (U-SQL) for external customers, and bringing Spark to Cosmos.

3.1 Designing for Scale

Figure 2 shows the core Cosmos architecture. The Cosmos frontend (CFE) layer is responsible for handling communication between the Cosmos cluster and the clients ❶. Each CFE server runs a front-end service, which performs authentication and authorization checks and provides interfaces for job submission and cluster management. If users are authorized to access data and submit jobs to the Virtual Cluster, the request is sent to Cosmos JobQueue Service (CJS) ❷. This service is responsible for scheduling a job on the backend servers based on resource availability and job priority. It also maintains the job's status. Once the job priority is satisfied and resources are available, the CJS sends the request to the SCOPE compiler service ❸, which carries out job compilation and optimization. During compilation, the SCOPE engine also communicates with Store Metadata service to get more information about data ❹.

The SCOPE compiler generates code for each operator in the SCOPE script and combines a series of operators into an execution unit or stage. The job can have many different stages in the job resulting in task parallelism. All the tasks in a single stage perform the same computation. The tasks can be scheduled separately and executed by a single back-end server on different partitions of the data, resulting in data parallelism of the stage. The compilation output of a script, therefore, consists of: (1) a graph definition file that enumerates all stages and the data flow relationships among them, (2) an unmanaged assembly, which contains the generated code, and (3) a managed assembly, which contains user assembly along with other system files. This package is then stored in Cosmos store and later downloaded on backend servers for job execution.

Once the compilation succeeds, the CJS sends a request to the YARN resource manager (RM) to schedule the SCOPE Job Manager (JM) on the Cosmos backend servers ❺❻. There is a large group of backend servers in the cluster that run a Node Manager (NM) service that is responsible for executing tasks as well as JM. SCOPE job execution is orchestrated by a JM, which is responsible for constructing the task graph using the definition sent by the compiler and the stream metadata ❼. Once it constructs the physical task graph it starts scheduling work across available resources in the cluster. To scheduling the task, JM fetches the health information of the servers ❽ and requests RM to schedule task on selected servers ❾. Once the tasks are scheduled ❿, created ⓫ and started ⓬, the JM also continuously monitors the status of each task, detects failures, and tries to recover from them without requiring to rerun the entire job. The JM periodically checkpoints its statistics to keep track of job execution progress. Task or vertex running on NM can read or write data to local or remote servers ⓭.

To ensure that all backend servers run normally, a resource health monitoring service (RMON) maintains state information for each server and continuously tracks their health via a heartbeat mechanism. In case of server failures, the RMON notifies every RM and JM in the cluster so that no new tasks are dispatched to the affected servers. As mentioned earlier, the NM service on each server is also responsible for managing containers that are ready to execute any assigned task. At execution time, the JM dispatches a request for resources to the RM to execute a task. The RM is responsible for finding a worker for task execution in the cluster. Once the RM sends a response to the JM with the worker information, the JM submits the request to schedule the task to the NM service running on the worker. Before executing the task on the worker, the NM is responsible for setting up the required execution environment which includes downloading required files to execute the task. Once the required files are downloaded, the NM launches the container to execute the task. The JM indicates whether the output of the task execution should be written as temp extents on the local server, or to the remote Cosmos store.

To avoid latency impact and to reduce the cost of operation, interactive queries follow different execution flow than batch/streaming queries. Once users send the request to CFE to execute an interactive query ❶, the request is directly sent to SCOPE compiler service

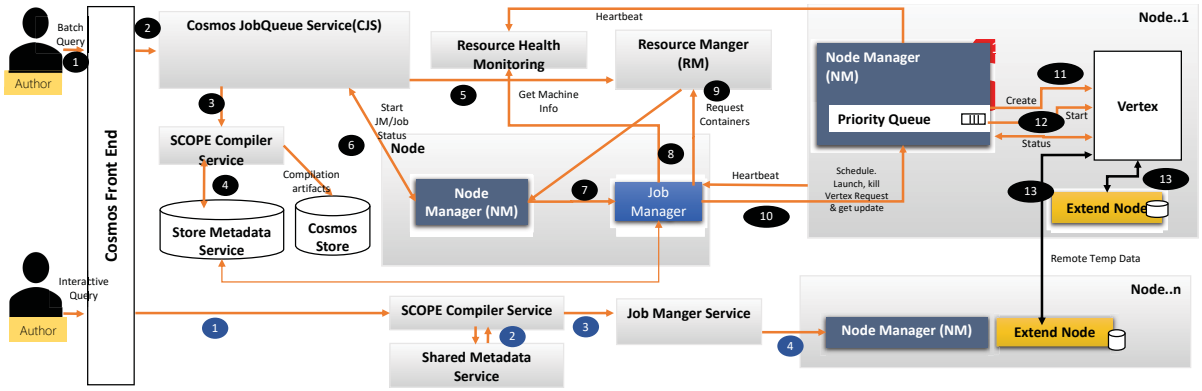


Figure 2: Cosmos Scope architecture in the last decade.

which carries out job compilation and optimization as explained earlier ❷. After compilation, SCOPE compiler service sends the job to Job Manger Service ❸ that starts orchestrating vertices for query execution ❹. Job Manager Service differs from JM in handling execution of multiple interactive queries compared to only a single query. On completion, the result are sent to users.

Cosmos runs on Autopilot [33] as the cluster management framework. Autopilot exposes a virtual data center abstraction that typically maps one-to-one to a physical data center, although there are exceptions. Within a virtual data center, Autopilot has the concept of virtual clusters, which are limited to 40,000 servers each. In 2013, multiple virtual clusters were stitched together to support Cosmos clusters larger than 40,000 servers. At that time, we scaled the size of Cosmos clusters up from 40,000 servers to 55,000 servers. Initially, virtual clusters that were stitched together were in same data center, however, later, the same approach was also used to connect virtual clusters in other data centers to that same Cosmos cluster. This unlocked support for Cosmos clusters to span multiple data centers. Today all Cosmos clusters span at least four data centers.

3.2 Compiler Architecture

An incoming SCOPE script goes through a series of transformations before it is executed. Firstly, the compiler parses the SCOPE script, unfolds views and macro directives, performs syntax and type checking, and resolves names. If the SCOPE script contains user define code like UDFs or UDOs in C#, Python, or Java, then the appropriate compiler is invoked to compile relevant code. The result of this step is an annotated abstract syntax tree, which is then passed to the query optimizer, that generates efficient execution plans. Below we describe the SCOPE compiler architecture in detail.

SCOPE is a declarative language, with C# and other language integration. Beyond relational operators, the entire expression language, the type system, extensibility model and even user references are all in terms of C# and .Net assemblies. This was a great win for Microsoft's internal .Net users, especially in terms of expressibility, but posed great challenges for the original version of the SCOPE compiler. As a result, the SCOPE compiler (as most compilers) attempted to have its own grammar, rules, binder, expression handling, and code-generation. Thus, a significant amount of developer effort was spent to match parity with the C# specification (even 1.0 for that matter), align C# semantics with the behavior and generation of the C# compiler itself, let alone trying to keep up with the ever-evolving versions of .Net and C# language additions (e.g., LINQ, lambdas, anonymous types, named parameters, generics).

Later, to gain agility, the compiler needed a major re-architecture, essentially splitting abstract syntax tree between two concepts: relational (algebra) and scalar (expression) services. Central to this investment was leveraging the .Net C# compiler services, code-named Roslyn [69], which was being actively developed in Community Technology Preview. These are the same C# services, parser, and semantic symbol binding that the C# compiler uses. This was a unique usage pattern that Roslyn was not designed for; originally optimized for a single app program on client-side. SCOPE was literally delegating to it for thousands of independent expression fragments, each with their own scoping rules, precedence, and relational context. Likewise, the Cosmos environments stretched Roslyn in terms of performance, scalability, and cluster-level reliability.

Consequently, the SCOPE compiler now has a modern and more natural integration with C#, not only from a user perspective but from the code and architecture itself. The compiler completely matches the C# specification and the actual C# compiler semantics bug-for-bug, with much less effort. SCOPE is now able to keep up with newer versions of .Net as well as C# language version at a much faster pace than we would have ever imagined, freeing up the team for other SCOPE language and programmability investments.

3.3 Heterogeneity

The Cosmos stack is designed to support a wide heterogeneity in workloads. Thanks to adoption across the whole of Microsoft, Cosmos needs to support workloads that are representative of multiple industry segments, including search engine (Bing), operating system (Windows), workplace productivity (Office), personal computing (Surface), gaming (Xbox), etc. To handle such diverse workloads, the Cosmos approach has been to provide a *one-size-fits-all experience*, as illustrated using a block diagram in Figure 3. We discuss the key heterogeneity characteristics below.

First, to make it easy for the customers to express their computations, Cosmos provides users a variety of programming language support, including declarative languages such as SCOPE and Spark

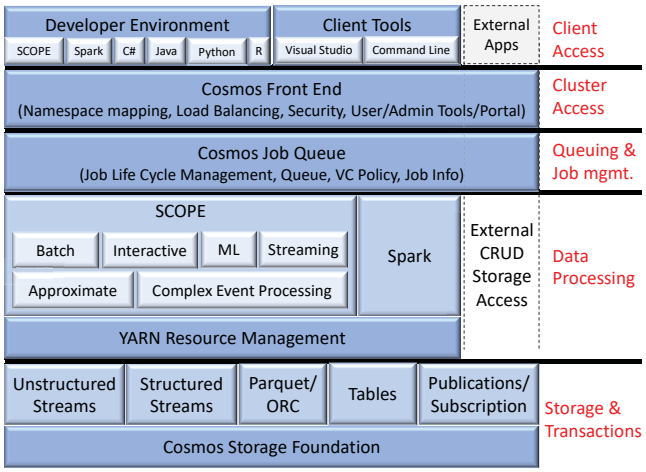


Figure 3: The Cosmos block diagram.

SQL, imperative languages such C# and Java, and data science languages such as Python and R. Furthermore, users can express all the above in simple data flow style computation and in the same job for better readability and maintainability. Cosmos also exposes a rich set of developer tools both in Visual Studio (coined ScopeStudio) and command line. Second, to enable different business scenarios, SCOPE supports different types of query processing, including batch, interactive, approximate, streaming, machine learning, and complex event processing. Third, SCOPE supports both structured and unstructured data processing. Likewise, multiple data formats, including both propriety and open-source such as Parquet, are supported. Cosmos further support other abstractions such as Tables and publish/subscribe APIs. Finally, considering the diverse workload mix inside Microsoft, we have come to the realization that it is not possible to fit every scenario using SCOPE. Therefore, we also support Spark query processing engine. Overall, the one-size-fits-all query processing experience in Cosmos covers diverse workloads, data formats, programming languages, and backend engines.

In addition to data processing using SCOPE or Spark, Cosmos also exposes storage access for CRUD (create, read, update, delete) operations by external applications. Baja, for example, implemented a log-structured file system on top of Cosmos to store many small files with frequent updates. As a result, it could support several new scenarios on Cosmos, including low latency storage, incremental processing, high volume data ingress/egress, random reads/writes with distributed transactions, and low latency event processing in pub-sub applications (sub-second latency, exactly-once processing, advanced load balancing, and fault tolerance). These are substantially different than traditional workloads in Cosmos.

Finally, to illustrate the heterogeneity in Cosmos workloads, Figures 4a—4c show the variance in the size of inputs processed, the end-to-end latencies, and the number of tasks per SCOPE job. We can see that the 50^{th} and 90^{th} percentiles vary by four orders of magnitude for the size of inputs, by almost two orders for the job latencies, and by more than two orders for the number of tasks, thus demonstrating a significant variance in SCOPE jobs. Furthermore, this variance could be up to an order of magnitude different on different production clusters, each dominated by workloads from different business units. The Cosmos infrastructure is designed to handle such heterogeneity in workloads.

3.4 Cluster Utilization

Cosmos has managed to maintain high cluster utilization over the years. To illustrate, Figure 5 shows the CPU utilization over the last decade. We can see the utilization hovering between 60% and 90%, with 70%+ in most of the years. There are several factors that have contributed to this consistently high utilization.

First, SCOPE engine can aggressively schedule vertices in a SCOPE job, even more than the maximum resources (called tokens [70]) allocated.This is done by leveraging unused resources from other jobs and repurposing them to opportunistically allocate vertices in more needy jobs [11]. In case a job requests its originally allocated resources back, then opportunistically allocated vertices are killed first. The SCOPE job manager also handles failures gracefully by retrying failed vertices multiple times and retrying them recursively up the dependency chain in case of multiple failures.

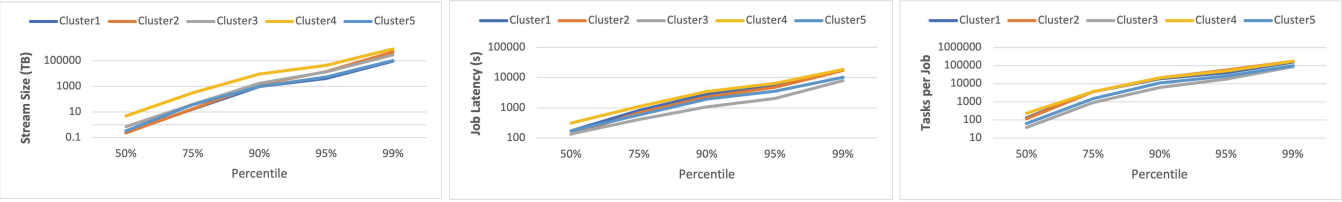
Second, a RMON (resource monitor) service in Cosmos captures the health information of every machine every ten seconds. Job managers can then query RMON for task utilization on different machines and push the load aggressively on those machines [11]. This means that there is no limit to the number of tasks that can run on a machine if resources are available, in contrast to scheduling a fixed number of containers per machine. RMON also has low operational overhead with only one instance needed per cluster.

Third, the user-defined operators (UDOs) in SCOPE could be CPU and memory-intensive. SCOPE allows such UDOs to borrow resources from other vertices running on the same machine. However, this is done with certain safeguards in place, e.g., kill the process beyond certain utilization, such as 20GB. Unfortunately, CPU and memory cannot be reclaimed by the original vertices.

Finally, the SCOPE engine exploits massive cluster resources for better performance. This includes better task and data parallelism using cost-based query optimization in SCOPE. As a result, expensive portions of the SCOPE job, can have large fan-outs, thus creating many tasks that could boost cluster utilization. Likewise, streaming SCOPE jobs have a large in-memory state to utilize the spare memory and avoid spilling to disk.

3.5 Developer Experience

The SCOPE engine supports a diverse workload which includes batch, interactive, streaming, and machine learning. Thus, the targeted users include developers, data engineers, business analysts, and data scientists. All of these play a critical role to influence an organization's big data strategy and technology choices and thus having a unified and easy-to-use experience is important. Common activities for these SCOPE users involve cleansing data, data aggregation and correlation, building process pipelines, tuning performance and scale, performing analysis queries, and visualizing results. Furthermore, in addition to authoring their code in the SCOPE declarative language, users rely on SCOPE extensions in C#, Python, and R to express their business logic, and therefore tools that increase productivity for development, debugging and tuning are important. SCOPE has support for state-of-the-art development



(a) Variane in stream sizes. **(b) Variance in job latency.** **(c) Variance in tasks per job.**

Figure 4: The variance in the batch SCOPE jobs in Cosmos.

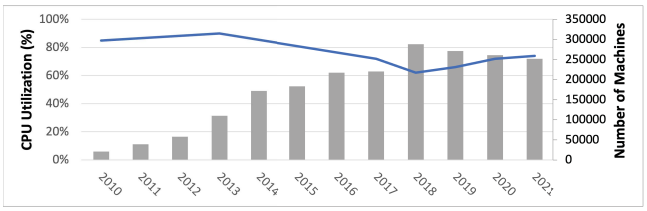


Figure 5: CPU utilization over the years in Cosmos.

tools in Visual Studio. This tool, called Scope Studio, provides a comprehensive authoring, monitoring, and debugging experience for all different kinds of workloads and language extensions. For authoring SCOPE jobs, Scope Studio provides Intellisense [56], which includes code completion to boost developer productivity. Once authored, Scope Studio allows users to run a SCOPE job locally or in a cluster. In both cases, they can see a visual representation of the job execution graph, along with relevant statistics (vertex counts, output row counts, latencies, etc.) as the execution progresses.

Bugs in SCOPE scripts are costly because they waste a lot of time and money, and resolving them is challenging due to the complex distributed execution environment and the difficulty in correlating runtimes issues with the highly optimized user scripts. Therefore, Scope Studio provides various features to make it easy to debug job failures and performance issues. For instance, static code analysis is a mature and powerful technique offering the first line of defense for SCOPE users during deployment. We provide a rule-based static code analysis tool, which is an analysis framework application shipped with Visual Studio. Scope Studio also helps in analyzing job execution, highlighting the bottlenecks, e.g., data skew, and giving recommendations to resolve them. During job execution, users can monitor the progress of the job, vertices, stages, as well as see various execution metrics like number of rows, data size, time taken, and memory consumed per vertex. Once the job has finished executing, users can analyze the critical path of the job, run what-if analysis for resource allocation, debug failed and successful vertices locally, or replay the entire job execution locally. Cosmos further stores telemetry from different services for future analysis to support a variety of offline debugging tasks, using Helios [62] to efficiently retrieve relevant logs for debugging failures. Another pain point for developers was to specify and allocate resources used for their queries. Autotoken [70] addresses this by predicting the peak usage of recurring jobs and allowing Cosmos to allocate only the required resources for users.

3.6 Efficiency Efforts

Cosmos has witnessed significant growth in usage over the years, from the number of customers (starting from Bing to almost every single business unit at Microsoft today), to the volume of data processed (from petabytes to exabytes today), to the amount of processing done (from tens of thousands of SCOPE jobs to hundreds of thousands of jobs today, across hundreds of thousands of machines). Even a single job can consume tens of petabytes of data and produce similar volumes of data by running millions of tasks in parallel. We considered several optimizations to handle this unprecedented scale. First, we *decoupled and disaggregated* the query processor from the storage and resource management components, thereby allowing different components in the Cosmos stack to scale independently [65]. Second, we scaled the data movement in the SCOPE query processor with better distribution and quasilinear complexity [20, 64, 86, 87]. This was crucial because data movement is often the most expensive step, and hence the bottleneck, in massive-scale data processing. Third, we introduced bubble scheduling [84] to divide the execution graph into sub-graphs called bubbles. Query operators within a bubble stream data between them. Fourth, we identified opportunities for reusing common computations in SCOPE jobs and built mechanisms for doing that [36, 37, 73]. Fifth, newer operator fusion techniques were proposed to identify interesting patterns in query plans and replace them with more efficient super operator implementations [45]. Sixth, there were opportunities for more aggressive data filtering [26, 40, 49] and skipping by creating replicas with heterogenous partitioning and sorting layouts [74]. And finally, SCOPE introduced sampling and approximate query processing techniques in Cosmos [39, 41, 83].

Efficiency improvements in the storage layer focused on improving the compression of data. We introduced a stronger default compression algorithm for data on-write that reduced data at rest by 30%. We also introduced recompression of data at rest, which allowed us to opportunistically consume spare CPU cycles to do more CPU-intensive compression that would otherwise incur higher latencies to do on-write. Finally, there were approaches to evaluate the amount of telemetry loss that could be tolerated [29].

3.7 Compliance Efforts

The General Data Protection Regulation (GDPR) is a European law that created new requirements for data retention and updatability in Cosmos. The granularity at which data needs to be deleted for GDPR meant that we had to add such deletion support to our Cosmos interface. To support GDPR in Cosmos, an append-only file system,

we added support for Bulk-Updatable Structured Streams (BUSS) which enabled customers to delete or update individual rows in their structured data. In 2018, because of the GDPR changes, we also saw a radical shift in the storage workload for Cosmos. GDPR changed the cost equation for Cosmos customers to store data, and we observed many customers electing to delete large portions of their cold Cosmos data. Deleting older data that was infrequently accessed increased average IOPS per byte at rest in Cosmos clusters. This meant supporting the same amount of network traffic and disk I/O with fewer hard drives, fewer servers, and fewer network links.

3.8 Azure Data Lake

By 2014, more and more external Microsoft customers were looking for offerings to process their own big data workloads. At that time, Microsoft offered HDInsight, a managed cluster form factor for the Hadoop eco-system. Since Cosmos was an easy-to-use platform as a service (PaaS) that had been successfully scaled to exabytes of data, we started developing external services called Azure Data Lake Analytics (ADLA) and Azure Data Lake Storage (ADLS) [65].

Azure Data Lake Storage (ADLS) [65] was a new software stack designed to support batch processing scenarios of Cosmos as well as other open-source workloads like Spark and HBase. Security and integration with Azure were key considerations in the ADLS design in a way that they had not been in Cosmos. The ADLS architecture added the Hekaton-based RSLHK metadata service to process high numbers of transactions needed for real-time applications. While migrating Cosmos storage onto ADLS, we kept the Cosmos EN and CSM but shifted many responsibilities of the CSM to the new PSM and Alki metadata services. This migration paved the way for many of the security and compliance improvements we are currently making in Cosmos. It also connected the Cosmos data lake to the Azure services allowing for seamless integration with Azure. For example, Cosmos customers can now access their Cosmos data from HDInsight, Azure Machine Learning, or an Azure VM. They can also use Azure Data Factory and Azure Event Grid to build complex workflows across cloud services.

Around the same time that we migrated the storage layer of Cosmos to the ADLS stack, we also migrated the resource management layer to YARN [80]. This migration, discussed at length in [18], enabled support for open-source engines other than SCOPE inside of Cosmos clusters. Over the past year, Cosmos users have started to use Spark inside Cosmos clusters. We expect the Cosmos workload to shift towards more open-source workloads in the future as Azure Synapse [57] makes the user experience for using open source frameworks on Cosmos easier through notebooks.

3.9 U-SQL

Considering the internal success of the SCOPE engine due to ease of use, scale, efficiency, and performance, and increased customer demand for a big data processing and analytics platform as a service, we decided to make it available to external customers. Since external customers expected a language closer to the familiar SQL, we "SQL-ified" the SCOPE language, and called the resulting language U-SQL [68]. U-SQL preserved SCOPE's strengths such as dataflow scripting, its tight C# integration for scalar expressions and functions as well as user-defined operators (UDOs), and kept the underlying runtime, optimizer, and execution framework of the SCOPE engine. However, U-SQL also introduced several new capabilities as described below.

U-SQL added a highly flexible fileset feature to read from and write to a large set of files. It also introduced SQL features that SCOPE (and even many SQL dialects) did not support at the time, such as PIVOT/UNPIVOT, complex data types such as MAP, UNION BY NAME to simplify unions of rowsets with partially overlapping schemas, handling of flexible rowset schemas (e.g., `SELECT *.Remove(col11, col450, col555) FROM @rowset`), etc. The addition of a meta-data service provided common database containers for logical objects such as tables, views, and computational objects such as table-valued functions, procedures, and a rich assembly management system that provides rich management capabilities for the user extensions (including support for user-code debugging) [53] not just in .NET/C# but also Python, R, and Java/Scala. Furthermore, U-SQL included predefined assemblies that provided built-in capabilities such as cognitive services for text and image analysis, JSON/XML support, and support for open-source structured data formats such as Parquet and ORC.

We released U-SQL as part of Azure Data Lake Analytics [50] in 2015 [51, 68]. The documentation [54] and the U-SQL release notes [52] provide a lot more details on its capabilities.

3.10 Spark Support in Cosmos

Cosmos customers were increasingly demanding open-source analytic engines such as Apache Spark [85]. This was motivated by market trends and the innovations in these engines, particularly the notebooks and libraries for data science and machine learning. Given the benefits of the Cosmos platform in terms of low cost, high scale, and the existing exabyte-sized data estate, we decided that it would make more sense to bring the open-source experience to Cosmos than to build out a completely different processing stack. The resource management and execution framework in Cosmos that is based on YARN is now processing both the SCOPE jobs as well as providing the resources and scheduling of Spark jobs and managing the Spark clusters that are being created for the Synapse Spark pools. Of course, we also need to provide interoperability between the most frequently used data formats. Therefore, as mentioned in the previous section, SCOPE has added support for Parquet and ORC, and we have now also added support for reading SCOPE's structured stream format in Spark by implementing a Spark data source connector. Furthermore, we extended SCOPE's meta-data service to be shared between SCOPE and Spark. This means that users can create a managed table in SCOPE that is backed by Parquet and read the same table in Spark, for example using SparkSQL. Since Cosmos customers possess a large set of .NET knowledge and libraries that they may want to reuse in Spark, support for .NET for Apache Spark [24] is now also included.

The Magpie [35] project explored automatically choosing the most efficient analytics engine for different tasks including Spark, SCOPE, and Azure SQL Data Warehouse. As part of this work, we looked at which jobs perform better on the Spark engine vs the SCOPE engine in Cosmos environments. We found that generally jobs with input sizes below 72 million rows perform better on Spark, while jobs with larger input sizes perform better on SCOPE.

4 LOOKING AHEAD: 2021 AND BEYOND

The greatest adventure is what lies ahead.
— J.R.R.Tolkien

In this section we describe our current work and planned future work to address the next set of challenges we see on the horizon for Cosmos. We expect our future efforts to focus on addressing fundamentals, efficiency, and the ways in which machine learning is changing the data systems landscape. The latter comes from both directions: "systems for ML" and "ML for systems". Our vision for ML support in Cosmos is to meet data scientists in their current programming languages and tooling ecosystem. On the other side of this coin, there has been a surge of recent literature on applying ML to improve data-intensive systems. Testing and productionizing these kinds of optimizations in a stable and debuggable way at scale is an important challenge for Cosmos going forward. Below we describe our work in progress to begin tackling these problems.

4.1 Security and Compliance

In the early days of Cosmos, the assumption was that all users are trusted. Users could read each other's data and execute arbitrary code on our backend servers. Over time, as the big data field has matured, we see high prioritization of security and compliance in big data systems [2, 8]. Recent efforts in Cosmos towards these ends include encrypting all data at rest, executing user-defined functions in secure containers, integrating Azure Active Directory for two-factor authentication, and moving to just-in-time (JIT) access for debugging production incidents. A challenge when enabling compliance and security in Cosmos is that we need to ensure that we maintain the efficiency of the system. We cannot increase the total cost of ownership or regress the performance of customer jobs.

Data encryption is crucial to our security efforts; however, we see several challenges with encryption at rest. These include maintaining the current compression ratios on encrypted data, encrypting existing data in the background invisible to customers, and sharing encrypted data between customers in a seamless way. Furthermore, we recompress data at rest in Cosmos for major efficiency advantages. However, this required us to build a new service that could be trusted to decrypt, recompress, and re-encrypt the data at rest. Finally, encrypted data is typically incompressible, and so we need to compress before encrypting, and that may mean compressing in different parts of the stack than before to ensure the data is encrypted end to end. We want to encrypt data as early as possible to reduce the number of places that decrypted data is around, and we also want to compress data as early as possible to minimize the amount of physical data we are moving around.

In summary, Security, compliance, and data governance have become major priorities for enterprise data lakes [2]. Ensuring security and compliance while also preserving high efficiency will be an ongoing focus for Cosmos over the next several years. The emergence of the cloud has created a new set of systems that Cosmos needs to integrate with, from moving data easily between cloud services to allowing cloud pipelines to coordinate via events.

4.2 Integrated with Azure Services

The traditional Cosmos approach has been to provide a one-size-fits-all experience, via an integrated ecosystem around a common SCOPE processing engine. However, this also complicates the system making it unpredictable, less performant, and unreliable. Considering the diverse workload mix inside Microsoft and the limitations seen in the SCOPE engine, we have come to the realization that it is not possible to fit all scenarios using SCOPE. Naturally, this means that we need to level up the integration of Cosmos with various other Azure services to enable newer end-to-end customer scenarios. To start with, we plan to invest in the integration of the SCOPE engine with various other data sources from which it can read and write data. While today SCOPE only supports persistent stores like ADLS and Cosmos, we plan to integrate with low latency stores and queuing services like Event Hub and/or Kafka. We also plan to integrate SCOPE with interactive engines like SQL on-demand and SQL DW to enable scenarios like dash-boarding, federated query execution, and reference data lookup.

4.3 Richer Analytical Models

As mentioned earlier, SCOPE supports complex and diverse workloads. SCOPE enables users to express business logic using SQL-like declarative relational language. Still, users often fall back to custom user code, i.e., user defined operators (UDO) or user define functions (UDF), in case they are not able to express the business logic in the high-level declarative language. In fact, around 90% of recurring jobs consist of at least one UDO inside it. This reflects the fact that the relational analytical model is not enough for users to express their business logic. Therefore, going forward we will be investing in adding a sequential analytical model and a temporal analytical model to support stateful operations of time-series analysis in the SCOPE language. Likewise, we need to add native support for graph and matrix operations in the SCOPE engine. MadLINQ [63] was an effort in this direction which introduced primitive scalable and efficient matrix computation in SCOPE. Similarly, there is initial work on supporting temporal queries in SCOPE [13–15].

4.4 Advanced Workloads

We have witnessed the rise of the data science community in recent years. Interestingly, in the enterprise setting, this new community of developers wants to apply newer libraries and algorithms for model learning while still operating on massive amounts of existing data. Furthermore, there is often a heavy tail of these data scientists who don't need heavy models or deep learning and SCOPE turns out to be a scalable and low-cost solution for their regression or classification tasks using popular libraries such as scikit-learn [61]. This is especially true for data that is partitioned and where multiple models can be trained in parallel, something that SCOPE is good at. For deep learning or other scenarios that need hardware offloading, we integrate with Azure Machine Learning (AML) to make the data processed in Cosmos available in AML via the Azure Data Lake Store (ADLS). The challenge, however, is to keep up with the rapidly updating machine learning libraries in Python. To carry on the momentum, we plan to continue our investments in supporting standardized machine learning frameworks for the above advanced workload scenarios.

Advanced workloads put pressure on various parts of the system, including different compute and store requirements from different workload scenarios, processing inputs from disparate data sources,

which could be data lake, SQL Server, or others, different resource allocation needs for high and low priority workloads, and workloads in newer regions, different user experiences that could include Notebooks or integration in newer IDEs such as VSCode, unlocking open-source workloads beyond Spark, making data science and machine learning easy, and providing data provenance applications. Putting all these together is key to taking the Cosmos integrated ecosystem to the next level of business and user expectations.

In the past Microsoft has looked into some of these advanced workloads for scalably processing queries on videos [47, 48], efficient data cleaning algorithms [4], efficiently searching over large text collections for Bing [67], at scale mapping IP address to geographical location [19] to mention a few.

4.5 Python Language Head

Python is the topmost programming language according to IEEE, the most popular language on Stack Overflow, and the second most popular language on GitHub. Furthermore, Python is becoming one of the most popular languages not only for data science but also among data engineers, and traditional developers. Despite the popularity, scaling Python to large datasets with good performance remains a challenge. This is in stark contrast to the world of database technologies, where for decades we invested in scalability, query optimization, and query processing. The work we will be doing in SCOPE will help bring together the ease of use and versatility of Python environments with highly performant query processing in the SCOPE engine. As a part of this work, we plan to expose the popular Python Dataframe API by mapping it to SCOPE operators under the hood. This will enable pushing large chunks of computation down to the scalable and efficient SCOPE engine, bringing the best of Python and scale to customers. Magpie [35] is an initial effort in this direction that exposes Pandas APIs to users while pushing expensive computation to scalable database engines.

4.6 Development Experience

Our goal is to have a seamless authoring, debugging, execution, and monitoring experience for data engineers and data scientists irrespective of the environment (local or cloud) on which analytical queries are executed. Traditionally, we have focused only on the standard development experience (offline authoring, debugging, and monitoring) and made little investments around the experience required to address the needs of data scientists and data engineers which is more along the lines of interactive development. As a next step, we would focus on interactive development in the language of choice, blending between cluster and local execution, recommending performance improvements while developing the script, identical development experience irrespective of the event to insight latency (batch, interactive, ML), and language support to make integration with other services seamless.

4.7 Small Job Optimization

Cosmos needs to support workloads that are representative of multiple industry segments, Bing, operating system (Windows), workplace productivity (Office), personal computing (Surface), gaming (Xbox), etc. Thus, Cosmos workloads are also complex and diverse. Furthermore, there is a continuous push from customers to enable them to do more data processing with the same amount of resources. Historically, SCOPE optimizer did an excellent job at optimizing large SCOPE jobs. However, recent analysis shows that 40% of the SCOPE workload now consists of jobs that run in less than 2 minutes. Therefore, we plan to invest in SCOPE (compiler, optimizer, code generation, operators, communication channel, and scheduler) to bring down latency for this class of small queries from 2 minutes to less than 30 seconds. This will be in addition to the continued investment in improving the large query segment of SCOPE to further bring down processing hours and query latencies.

4.8 Workload Optimization

The rise of managed data services like Cosmos has made it extremely difficult for users to manage their workloads, given the lack of control as well as expertise on a sophisticated data processing stack. At the same time, improving operational efficiency and reducing costs have become paramount in modern cloud environments to make the infrastructure viable. Consequently, we are building several features to analyze and optimize end-to-end customer workloads. An example is to right-size the resource allocation for SCOPE jobs, both in terms of the degree of parallelism (maximum concurrent containers) as well the memory used in each of the containers [7, 70, 72]. Another example is to insert checkpoints such that the cost to restart failed jobs is minimized. Another example is to help customers detect the right physical layouts for their input files [60] or to collect statistics [59] on those files to create better query plans over those inputs. Or, using past workloads for better scheduling decisions [9, 16, 28, 31]. All of these are hard for customers to figure out, given the complexity and scale of the workloads. Therefore, we are investing in a common workload optimization platform that could be leveraged across customers to optimize their overall workload and to save costs.

4.9 ML-for-Systems

Applying machine learning (ML) techniques to optimize data-intensive systems has received significant attention in recent years [32, 42, 79]. We have investigated the use of machine learning for auto-tuning many layers of the Cosmos software stack. For example, the SCOPE query optimizer estimates are often far from ideal, and so we applied machine learning to learn fine-grained models [38] to improve estimates such as cardinality [81] and cost models [72], and others. Likewise, we also apply ML to tune the underlying platform, including things such as YARN parameters, power provisioning, and server resource ratios in Cosmos [88]. And finally, there are efforts to improve incident management in Cosmos by predicting the causes of slowdown in recurring jobs in case of an incident [71]. All the above examples have shown great potential to improve the current state by leveraging and learning from large volumes of past workloads, and therefore we will continue to invest in them.

5 THE MODERN ANALYTICS LANDSCAPE

It's always nice to see familiar faces.
— Kyle Walker

The modern data analytics landscape is evolving rapidly [1, 8]. Even the expectations from a data lake are quickly going beyond daily batch jobs and multi-minute ad-hoc queries, which were some

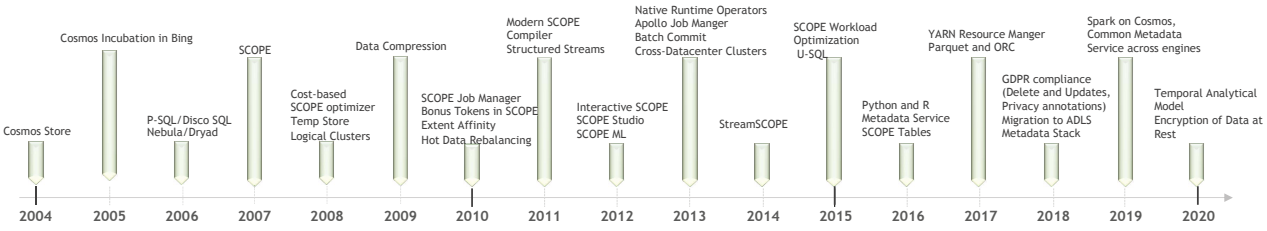


Figure 6: Major Cosmos timelines over the years.

of the original scenarios in Cosmos. Below we discuss and reflect on some of the newer industry trends and relate them to Cosmos.

From SQL to NoSQL, and all the way back. We have witnessed the rise and fall of NoSQL systems over the last couple of decades. The need for massive scale-out, flexible schemas, and arbitrary user code lead to this new breed of systems. And indeed, close integration with C# and the ability to express complex business logic using a variety of user defined operators were some of the key reasons for the popularity of SCOPE and Cosmos. At the same time, there has been a continuous shift in this new breed of systems towards more declarative semantics. For SCOPE, this meant building a sophisticated compiler using Roslyn, introducing newer operators such as temporal operators, and explicitly SQL-ifying SCOPE into U-SQL for external customers. This paralleled development in other systems such as Hadoop and Spark that evolved into Hive and Spark SQL as their popular declarative query interfaces. Thus, while the NoSQL movement was instrumental in rallying a large developer audience, not necessarily database experts, for scalable data processing, the eventual shift towards declarative interfaces leads to better developer productivity, system performance, and maintainability of enterprise codebases.

From Hadoop to Spark ecosystem. Big data processing systems tend to build an ecosystem for supporting a wide variety of analytics applications. These systems incur large investments and so users want to get the most out of them. Furthermore, it is impractical to move large volumes of data around into different systems for different applications. In the open-source community, it started with the Hadoop ecosystem to support a plethora of data processing capabilities on top of a common Hadoop Distributed File System. Over time, Spark has developed a similar ecosystem with better interactivity and developer productivity. The Cosmos ecosystem has seen a similar journey where it grew into a one stop shop for all analytical processing needs within Microsoft.

SCOPE versus Spark SQL. SCOPE has been the query processing workhorse in Cosmos, however, increasingly Spark has become highly attractive due to better performance over interactive queries, better developer productivity with Notebooks, and more comprehensive features for building data science applications. As a result, we have concluded that no one system fits all scenarios, and Cosmos now offers Spark along with SCOPE. Interestingly, SCOPE still offers lower operational costs due to better cluster utilization and scale-out, as well as tighter C# integration, which is a big advantage for Microsoft developers. As a result, SCOPE has a niche for exabyte scale data ingress, cooking, and transformations.

Lake-first and cloud-first. Data lakes and data warehousing are increasingly more tightly coupled for an end-to-end experience, including data ingestion, preparation, analysis, reporting, interactive querying, and exploration. This has been coined *LakeHouse* by Databricks [5, 6], while Dremio [22] positions as a *lake engine* that can provide the above processing on top of the lake. Snowflake [75], on the other hand, has built a cloud-native data warehousing architecture, that decouples store and compute and provides unlimited scaling capabilities similar to a data lake. Cosmos provides a similar storage disaggregation (ADLS) that can scale independently, and it supports warehousing either through an interactive engine, called *iSCOPE*, or through integration with Microsoft's warehousing and reporting tools such as Azure Synapse SQL and PowerBI.

Workflow orchestrators, low-code/no-code. Data analytics is now accompanied by workflow orchestration tools to help stitch together, monitor, and maintain end-to-end workflows (e.g., Airflow, Luigi, Dagster, Prefect, Metaflow, Dask). Along similar lines, several workflow engines have been built by Cosmos customer teams for their needs (e.g., Xflow, Sangam). Azure Data Factory, which has become the standard for workflow orchestration on Azure, is now also integrated with Cosmos. There is a further push for low-code/no-code solutions for business analysts and several internal tools, such as Aether, provide a low-code and no-code experience.

6 CONCLUSION

In this paper, we discuss the past, present, and future of Cosmos, the exabyte-scale big data processing platform at Microsoft. Figure 6 summarizes our journey by showing the major timelines in the past 16 years, starting all the way to an early version of Cosmos store, to developments in SCOPE and cost-based query optimization, to support for newer workloads via SCOPE ML and StreamSCOPE, to addressing modern-day challenges such as GDPR, compliance, and compatibility with open-source software. Starting all the way from the origins, we saw how customer needs in Cosmos have changed over time, how the core architecture got stabilized over numerous iterations, what newer advancements and workload types kept coming, how we still have a long way and newer challenges ahead of us, and how Cosmos continues to be relevant and competitive in the modern data analytics parlance.

ACKNOWLEDGEMENTS

We would like to thank everyone who have helped build Cosmos over the years, and all who continue to build it today.

REFERENCES

[1] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A Bernstein, Michael J Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J Franklin, et al. 2016. The Beckman report on database research. *Commun. ACM* 59, 2 (2016), 92–99.

[2] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Kostantinos Karanasos, Subru Krishnan, Brian Kroth, et al. 2019. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. *arXiv preprint arXiv:1909.00084* (2019).

[3] Amazon. 2017. Amazon Athena. https://docs.amazonaws.cn/en_us/athena/latest/APIReference/athena-api.pdf.

[4] Arvind Arasu, Surajit Chaudhuri, Zhimin Chen, Kris Ganjam, Raghav Kaushik, and Vivek Narasayya. 2012. Experiences with using data cleaning technology for bing services. *Data Engineering Bulletin* (2012).

[5] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[6] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. [n.d.]. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. ([n. d.]).

[7] Malay Bag, Alekh Jindal, and Hiren Patel. 2020. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[8] Peter Bailis, Juliana Freire, Magda Balazinska, Raghu Ramakrishnan, Joseph M Hellerstein, Xin Luna Dong, and Michael Stonebraker. 2020. Winds from seattle: database research directions. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3516–3516.

[9] Peter Bodík, Ishai Menache, Joseph Naor, and Jonathan Yaniv. 2014. Deadline-aware scheduling of big-data processing jobs. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. 211–213.

[10] W.J. Bolosky, D. Bradshaw, R.B. Haagens, N.P. Kusters, and P. Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *2011 In Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation*. 141–154.

[11] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 285–300.

[12] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 143–157.

[13] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, and James F Terwilliger. 2015. Trill: Engineering a Library for Diverse Analytics. *IEEE Data Eng. Bull.* 38, 4 (2015), 51–60.

[15] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. 2012. Temporal analytics on big data for web advertising. In *2012 IEEE 28th international conference on data engineering*. IEEE, 90–101.

[16] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 1205–1223.

[17] Gavin Clarke. 2008. *Microsoft's Red-Dog cloud turns Azure.* Retrieved January 22, 2021 from https://www.theregister.com/2008/10/27/microsoft_amazon/

[18] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. 2019. Hydra: a federated resource manager for data-center scale analytics. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 177–192.

[19] Ovidiu Dan, Vaibhav Parikh, and Brian D Davison. 2016. Improving IP geolocation using query logs. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 347–356.

[20] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. Clusterjoin: A similarity joins framework using map-reduce. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1059–1070.

[21] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).

[22] Dremio. 2021. Dremio. https://www.dremio.com/data-lake/.

[23] Mary Jo Foley. 2009. *Red Dog: Five questions with Microsoft mystery man Dave Cutler.* Retrieved January 22, 2021 from https://www.zdnet.com/article/red-dog-five-questions-with-microsoft-mystery-man-dave-cutler/

[24] .NET Foundation. 2020. .NET for Apache Spark. https://github.com/dotnet/spark.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[26] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony Rowstron. 2013. Rhea: automatic filtering for unstructured cloud storage. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 343–355.

[27] Google. 2015. Google Cloud Dataflow. https://cloud.google.com/dataflow/.

[28] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 81–97.

[29] Jayant Gupchup, Yasaman Hosseinkashi, Pavel Dmitriev, Daniel Schneider, Ross Cutler, Andrei Jefremov, and Martin Ellis. 2018. Trustworthy Experimentation Under Telemetry Loss. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 387–396.

[30] Apache Hadoop. 2005. https://hadoop.apache.org.

[31] Yuxiong He, Jie Liu, and Hongyang Sun. 2011. Scheduling functionally heterogeneous systems with utilization balancing. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 1187–1198.

[32] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*, Vol. 11. 261–272.

[33] Michael Isard. 2007. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review* 41, 2 (2007), 60–67.

[34] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[35] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. [n.d.]. Magpie: Python at Speed and Scale using Cloud Backends. ([n. d.]).

[36] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[37] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung 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*. 191–203.

[38] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2021. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. In *ICDE*.

[39] Srikanth Kandula, Kukjin Lee, Surajit Chaudhuri, and Marc Friedman. 2019. Experiences with approximating queries in Microsoft's production big-data clusters. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2131–2142.

[40] S. Kandula, L. Orr, and S. Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. In *Proceedings of the VLDB Endowment, 13(3)*. 252–265.

[41] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*. 631–646.

[42] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. In *CIDR*.

[43] SPT Krishnan and Jose L Ugia Gonzalez. 2015. Google cloud dataflow. In *Building Your Next Big Thing with Google Cloud Platform*. Springer, 255–275.

[44] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. 2012. The Unified Logging Infrastructure for Data Analytics at Twitter. *Proceedings of the VLDB Endowment* 5, 12 (2012).

[45] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating super-operators in big-data query optimizers. *Proceedings of the VLDB Endowment* 13, 3 (2019), 348–361.

[46] Jimmy Lin and Dmitriy Ryaboy. 2013. Scaling big data mining infrastructure: the twitter experience. *Acm SIGKDD Explorations Newsletter* 14, 2 (2013), 6–19.

[47] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. 2016. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 57–70.

[48] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. 2016. Visflow: a relational platform for efficient large-scale video analytics. In *ACM Symposium on Cloud Computing (SoCC)*.

[49] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*. 1493–1508.

[50] Microsoft. 2015. Azure Data Lake. https://azure.github.io/AzureDataLake/.

[51] Microsoft. 2016. U-SQL. http://usql.io.

[52] Microsoft. 2016. U-SQL Release Notes. https://github.com/Azure/AzureDataLake/tree/master/docs/Release_Notes.

[53] Microsoft. 2017. U-SQL Data Definition Language. https://docs.microsoft.com/en-us/u-sql/data-definition-language-ddl-statements.

[54] Microsoft. 2017. U-SQL Language Reference. https://docs.microsoft.com/en-us/u-sql/.

[55] Microsoft. 2018. Azure RSL. https://github.com/Azure/RSL.

[56] Microsoft. 2018. IntelliSense. https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2019.

[57] Microsoft. 2021. Azure Synapse Analytics. https://azure.microsoft.com/en-in/services/synapse-analytics/.

[58] Pulkit A Misra, María F Borge, Íñigo Goiri, Alvin R Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[59] Azade Nazi, Bolin Ding, Vivek Narasayya, and Surajit Chaudhuri. 2018. Efficient estimation of inclusion coefficient using hyperloglog sketches. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1097–1109.

[60] Rimma Nehme and Nicolas Bruno. 2011. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1137–1148.

[61] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[62] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, et al. 2020. Helios: hyperscale indexing for the cloud & edge. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3231–3244.

[63] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. 2012. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*. 197–210.

[64] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. 2019. Hyper dimension shuffle: Efficient data repartition at petabyte scale in scope. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1113–1125.

[65] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 51–63.

[66] W.D. Ramsey and R.I. Chaiken. U.S. Patent 7,840,585, 2010. DISCOSQL: distributed processing of structured queries.

[67] Knut Magne Risvik, Trishul Chilimbi, Henry Tan, Karthik Kalyanaraman, and Chris Anderson. 2013. Maguro, a system for indexing and searching over very large text collections. In *Proceedings of the sixth ACM international conference on Web search and data mining*. 727–736.

[68] Michael Rys. 2015. Introducing U-SQL – A Language that makes Big Data Processing Easy. https://devblogs.microsoft.com/visualstudio/introducing-u-sql-a-language-that-makes-big-data-processing-easy/.

[69] Mehrdad Saadatmand. 2017. Towards Automating Integration Testing of. NET Applications using Roslyn. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 573–574.

[70] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. Autotoken: Predicting peak parallelism for big data analytics at microsoft. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3326–3339.

[71] Liqun Shao, Yiwen Zhu, Siqi Liu, Abhiram Eswaran, Kristin Lieber, Janhavi Mahajan, Minsoo Thigpen, Sudhir Darbha, Subru Krishnan, Soundar Srinivasan, et al. 2019. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-based Platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 441–452.

[72] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.

[73] Yasin N Silva, Paul-Ake Larson, and Jingren Zhou. 2012. Exploiting common subexpressions for cloud query processing. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1337–1348.

[74] Muthian Sivathanu, Midhul Vuppalapati, Bhargav S Gulavani, Kaushik Rajan, Jyoti Leeka, Jayashree Mohan, and Piyus Kedia. 2019. Instalytics: Cluster filesystem co-design for big-data analytics. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 235–248.

[75] Snowflake. 2021. Snowflake Data Cloud. https://www.snowflake.com/.

[76] Roshan Sumbaly, Jay Kreps, and Sam Shah. 2013. The big data ecosystem at linkedin. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1125–1134.

[77] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 1013–1020.

[78] New York Times. 2009. A Deluge of Data Shapes a New Era in Computing. https://cacm.acm.org/news/54396-a-deluge-of-data-shapes-a-new-era-in-computing/fulltext.

[79] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024. https://db.cs.cmu.edu/papers/2017/p1009-van-aken.pdf

[80] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[81] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.

[82] Ming-Chuan Wu, Jingren Zhou, Nicolas Bruno, Yu Zhang, and Jon Fowler. 2012. Scope playback: self-validation in the cloud. In *Proceedings of the Fifth International Workshop on Testing Database Systems*. 1–6.

[83] Ying Yan, Liang Jeff Chen, and Zheng Zhang. 2014. Error-bounded sampling for analytics on big sparse data. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1508–1519.

[84] Zhicheng Yint, Jin Sun, Ming Li, Jaliya Ekanayake, Haibo Lin, Marc Friedman, José A Blakeley, Clemens Szyperski, and Nikhil R Devanur. 2018. Bubble execution: resource-aware reliable analytics at cloud scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 746–758.

[85] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[86] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 13–24.

[87] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 1060–1071.

[88] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens, Sarvesh Sakalanaga, Sudhir Darbha, Minu Iyer, Ankita Agarwal, and Carlo Curino. [n.d.]. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of data*.

[89] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. In *1977 IEEE Transactions on information theory, 23(3)*. 337–343.