Automatically Indexing Millions of Databases in Microsoft Azure SQL Database

Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic,
Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic,
Gaoxiang Xu, Surajit Chaudhuri
Microsoft Corporation

ABSTRACT

An appropriate set of indexes can result in orders of magnitude better query performance. Index management is a challenging task even for expert human administrators. Fully automating this process is of significant value. We describe the challenges, architecture, design choices, implementation, and learnings from building an industrial-strength auto-indexing service for Microsoft Azure SQL Database, a relational database service. Our service has been generally available for more than two years, generating index recommendations for every database in Azure SQL Database, automatically implementing them for a large fraction, and significantly improving performance of hundreds of thousands of databases. We also share our experience from experimentation at scale with production databases which gives us confidence in our index recommendation quality for complex real applications.

ACM Reference Format:
Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of ACM SIGMOD conference (SIGMOD'19).* ACM, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

1.1 Motivation

A relational database with an appropriate set of indexes can have orders of magnitude better performance and lower resource consumption. Choosing good indexes is challenging even for expert Database Administrators (DBA) since it is workload, schema, and data dependent. Decades of research [2, 9–11, 13, 14, 18, 40, 46] and many commercial tools [2, 14, 46] have helped DBAs search the complex search space of alternative indexes (and other physical structures such as materialized views and partitioning). However, a DBA still drives this tuning process and is responsible for several important tasks, such as: (i) identifying a representative workload; (ii) analyzing the database without impacting production instances; (iii) implementing index changes; (iv) ensuring these actions do not adversely affect query performance; and (v) continuously tuning the database as the workload drifts and the data distributions change.

Cloud database services, such as Microsoft Azure SQL Database, automate several important tasks such as provisioning, operating system and database software upgrades, high availability, backups etc, thus reducing the total cost of ownership (TCO). However, in most cases, performance tuning, e.g., index management, remains the user's responsibility. Offering existing on-premises tuning tools hosted as a cloud service still leaves a significant burden on the users, which is daunting for many users of cloud databases who are application developers lacking expert DBA skills.

Another pattern we commonly observe is Software-as-a-Service (SaaS) vendors and Cloud Software Vendors (CSV) deploying hundreds to thousands of databases for customers of their applications. Managing such a huge pool of databases is a formidable task even for expert DBAs, where individual DB instances can have different schema, queries, and data distributions [24, 33, 39]. Therefore, being able to *fully automate indexing*, i.e., not only identifying the appropriate indexes, but also automating the above-mentioned steps crucial for holistic index lifecycle management, is a significant step in the cloud's promise of reducing the TCO.

Azure SQL Database is a fully-managed relational database-as-a-service with *built-in intelligence* [6]. Query performance is tuned through services such as automated plan correction [4], automated indexing [5], and intelligent insights [19]; here we focus on auto-indexing. Existing approaches that rely on human intervention for the critical steps in indexing do not scale for Azure SQL Database automating indexing for its millions of databases. Therefore, several years back, we embarked on the journey to build the first industrial-strength fully-automated indexing solution for Azure SQL Database.

SIGMOD'19, June 2019, Amsterdam, NL
© 2019 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1.2 Challenges

The *first* challenge is to scale the auto-indexing service to all databases in Azure SQL Database while meeting the statutory and compliance requirements. Azure has more than 50 regions worldwide spread over more than 140 countries [7]. The auto-indexing service must operate with minimal or no human intervention to achieve this scale. In addition to ensuring the proper health and operation of this service, we need to be able to debug the *quality* of the recommendations as the workloads and databases change. Moreover, the database server bits are frequently upgraded, our service itself goes through upgrades, and we must tolerate a wide variety of software and hardware failures we see at Azure-scale. Unlike a DBA tuning a database who can examine the queries or understand the applications, due to compliance reasons, all of the above actions must be performed without the engineers having knowledge of the application, its workload, or data distributions, which further amplifies the challenge!

The *second* challenge is to automate generating the several critical inputs to the index tuner needed for high quality recommendations. Unlike a DBA who can use additional application context, we need to automatically identify a workload to tune indexes for and specify other tuning constraints, e.g., the types of indexes to recommend, storage budget, or maximum number of indexes, etc.

The *third* challenge is that the state-of-the-art index recommenders [9, 14, 31, 46] rely on the query optimizer's cost estimates to determine the benefits of new index configurations. However, users care about actual query execution cost (e.g., query's CPU time). Due to well-known limitations of the query optimizer's estimates [20, 21], there are many instances where an index, which is estimated to improve query performance based on the optimizer's cost, makes query execution costs (or performance) worse once implemented [8, 16, 17]. Having the ability to ideally avoid or at least quickly detect and correct query performance regressions due to index changes is crucial in a production setting.

The *fourth* challenge is to ensure auto-indexing, which is a built-in service, does not significantly impact normal application workload and interfere or disrupt application processes. This has two facets: (*a*) low resource footprint; and (*b*) not blocking user operations. There are databases in Azure SQL Database which only have less than a CPU core allocated. Therefore, the service must be able to operate with low resource footprint. To support our user's 24×7 business continuity, indexing must not block user's workload or upgrades to the application or database schema.

1.3 Approach and Contributions

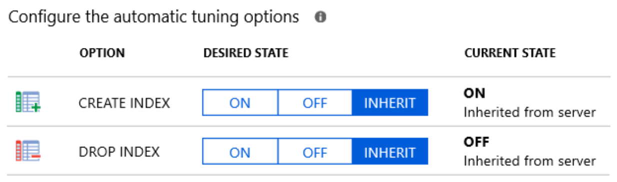
We describe how we overcame the above-mentioned challenges and built Azure SQL Database's auto-indexing service. We explain the user-facing auto-indexing offering (Section 2), where we expose several controls for expert users to configure the service and provide transparency of the indexes implemented and their impact on performance. We identify the architecture and the key components essential to *fully-automate recommendation and implementation* of indexes (Section 3). These components are: (*a*) *control plane*: the backbone of our automation that drives index lifecycle management and coordinates the other components in the system (Section 4); (*b*) *index recommender*: analyzes the workload to identify an appropriate set of indexes to create or drop (Section 5); (*c*) *validator*: a novel component that analyzes the impact of implementing the indexes, detects if performance has improved or regressed, and takes corrective action if it detects a significant regression (Section 6).

The control plane is crucial to scale the service to millions of databases in Azure SQL Database. It is a fault-tolerant service that automatically invokes the recommender, implements indexes (with user's permission), validates them, monitors the service health, detects issues, and wherever possible takes corrective actions. To meet the compliance requirements, any system component that accesses sensitive customer data, e.g., query text or data distributions, runs within Azure's compliance boundaries that is guarded by multiple levels of security, authentication, and auditing. Health signals of the service and debugging is primarily through anonymized and aggregated data.

The index recommender analyzes query execution statistics to automatically identify inputs, such as a workload to tune. To generate high quality index recommendations, we adapt and extend two building blocks for index recommendations in SQL Server: (*a*) Missing Indexes [34]; and (b) Database Engine Tuning Advisor (DTA) [2, 10]. These approaches provide complementary benefits which we together leverage to support the huge diversity requirements in Azure SQL Database, e.g., tuning databases with very different workload complexity and available resources [28].

The validator compares the execution costs before and after implementing an index change, detects large regressions, and takes corrective actions (e.g., automatically reverting the created index) to fix such regressions. Thus, we can tolerate the shortcoming of the index recommender relying on the optimizer's estimates and automatically implement recommendations while limiting impact to production workloads.

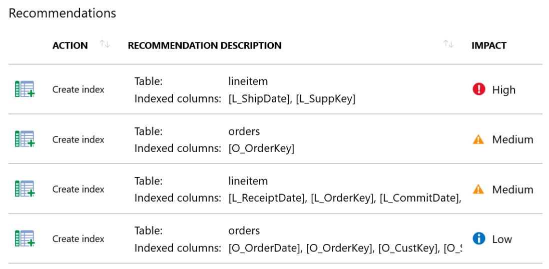
To ensure our automated actions do not adversely affect application performance, we carefully select the automated actions and build necessary mechanisms to prevent or minimize disruptions. For instance, we only support online indexing operations that does not block queries, drop auto-created indexes whenever it conflicts with an application's actions such as schema change or partition switching [44], minimize metadata contention using different locking levels [43], and



Configure the automatic tuning options

OPTION	DESIRED STATE	CURRENT STATE
CREATE INDEX	ON / OFF / [✓ INHERIT]	ON Inherited from server
DROP INDEX	ON / OFF / [✓ INHERIT]	OFF Inherited from server

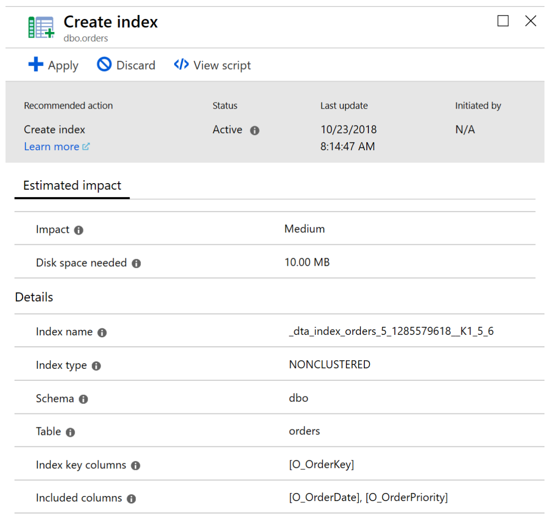
Figure 1: A snapshot of the UI that customers use in Azure portal to configure the auto-indexing service.



Recommendations

ACTION	RECOMMENDATION DESCRIPTION	IMPACT
Create index	Table: lineitem Indexed columns: [L_ShipDate], [L_SuppKey]	High
Create index	Table: orders Indexed columns: [O_OrderKey]	Medium
Create index	Table: lineitem Indexed columns: [L_ReceiptDate], [L_OrderKey], [L_CommitDate],	Medium
Create index	Table: orders Indexed columns: [O_OrderDate], [O_OrderKey], [O_CustKey], [O_	Low

Figure 2: A snapshot of the UI where customers see the index recommendations/actions.



Create index
dbo.orders

Recommended action	Status	Last update	Initiated by
Create index Learn more	Active	10/23/2018 8:14:47 AM	N/A

Estimated impact

Impact	Medium
Disk space needed	10.00 MB

Details

Index name	_dta_index_orders_5_1285579618__K1_5_6
Index type	NONCLUSTERED
Schema	dbo
Table	orders
Index key columns	[O_OrderKey]
Included columns	[O_OrderDate], [O_OrderPriority]

Figure 3: A snapshot of the UI where customers see the recommendation details.

use resource governance to isolate resources for tuning and index builds from concurrent workload [15, 25].

Azure SQL Database imposes few restrictions on the types of applications and workloads that customers deploy. Since we take the responsibility to fully automate indexing, we must maintain high quality of index recommendations in spite of this huge diversity in schema, workloads, and data distributions. To build confidence on our recommendation quality, we leverage the unique opportunity in Azure SQL Database to *experiment at scale* with production workloads. Since index changes can have significant performance impact, such experimentation is not possible on a database's primary replica. We create *B-instances*, which are independent copies of the database seeded from a snapshot of the primary copy where the real application's workload is forwarded and replayed in parallel to the primary copy. We make physical design changes or enable new features on these B-instances without impacting the primary workload, allowing us to experiment with different index configurations and measure their impact on execution (Section 7). While such experimentation has significantly helped the auto-indexing service, it is an orthogonal component which is used by many other services in Azure SQL Database.

Our service launched in preview in July 2015 and has been generally available since January 2016. Our current offering manages non-clustered (secondary) B+ tree indexes, has successfully implemented and validated millions of indexes, and significantly improved performance of hundreds of thousands of databases. At the time of writing, Azure SQL Database's automated indexing service is, to the best of our knowledge, the only fully-automated solution indexing millions of databases. We summarize our major learnings from operationalizing this service and the feedback from hundreds of customers (Section 8). We also summarize related work (Section 9) and conclude the paper discussing some hard challenges ahead of us (Section 10).

2 AUTO-INDEXING OFFERING

Azure SQL Database exposes a hierarchy of logical servers and logical database endpoints where a logical server can have one or more logical databases. To fully automate indexing, we eliminate the *need* for human inputs. However, we allow human intervention and provide controls to configure the service and selectively turn off features. We expose such controls through the Azure management portal, REST API for management, and T-SQL API [5]. Figure 1 provides a snapshot of the UI where users can configure the auto-indexing service for each database using Azure Portal. A user can also specify a setting for the logical server which every database on the server can inherit. The service automatically generates recommendations to create and drop indexes. These settings control whether the recommendations are automatically implemented and validated on the user's behalf. There are also defaults for cases where the user decides not to exercise this control. In Figure 1, the database is configured (see Current State column) to automatically create indexes, but only recommend indexes to drop (both settings inherited from the logical server).

To maintain transparency, we always show a list of current recommendations and the history of indexing actions. Figure 2 provides a summary of the different recommendations

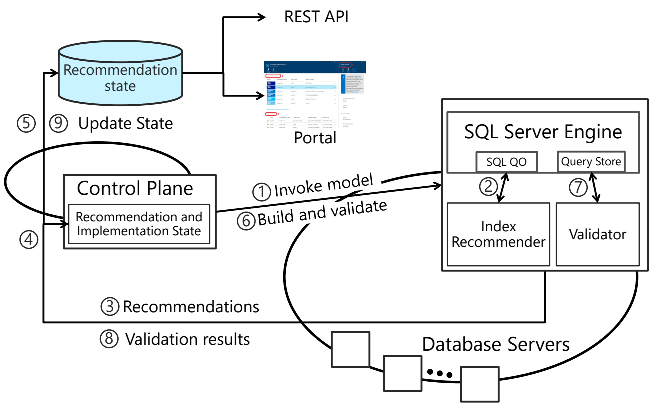


Figure 4: Conceptual architecture of auto-indexing service within a single Azure region.

for a demo database. Each item descries the index (i.e., the table, key columns, and included columns), and its estimated impact. A detailed view (e.g., see Figure 3) shows additional information e.g., index size. A list of SQL statements which will be impacted once the index is implemented is also exposed. If the database is configured for auto-implementation, then the system will execute these operations and validate them. If auto-implementation is off, then the user has an option of analyzing the recommendations and selectively applying them. The user can manually specify the system to apply a recommendation which are validated by the system. A user also has the option to copy the details and apply the recommendation themselves (e.g., through a schema management tool). In this case the user is responsible to validate them. For every action implemented by the system, a history view shows the state of such actions, e.g., successfully validated, reverted etc, and summarize the actual query execution costs before and after implementation.

3 ARCHITECTURAL OVERVIEW

We identify the key components essential to fully automate indexing at the scale of Azure SQL Database. Figure 4 provides a conceptual architecture of how these loosely-coupled components work together. Each Azure region comprises one or more database clusters spread over one or more data centers. There is one instance of the auto-indexing service managing all clusters and databases in a region. This per-region design meets the data sovereignty and compliance requirements and avoids data flow across regions.

Figure 4 numbers the different steps in the typical workflow of states when auto-indexing a database. The Control Plane is the brain of the system, designed as a highly-available service that manages the fault-tolerant state machine of auto-indexing for each database (see Section 4) and ensures the health of the auto-indexing service. The different system components are decentralized and communicate through asynchronous RPCs and events emitted through Azure SQL Database telemetry. The control plane invokes the Index Recommender to analyze the workload and generate index recommendations (see Section 5). Depending on whether the database is enabled to automatically implement the indexes or the user invokes creation of indexes through the system, the control plane implements the actions and invokes the Validator (see Section 6).

There are two SQL Server components that we rely on to build the auto-indexing service. *First* is the SQL Server query optimizer. It generates the missing index candidates per query [34]. We also use the optimizer's "what-if" API to simulate hypothetical index configurations [11]. *Second* is Query Store [29] which persistently tracks the query text, history of query plans, and a variety of execution statistics (e.g., CPU time, duration, logical and physical I/O, etc.) aggregated over time intervals. The Index Recommender and Validator relies on Query Store for several critical operations.

4 CONTROL PLANE

The control plane orchestrates the different components of the auto-indexing service. Every database in Azure SQL Database is independently indexed. Therefore, conceptually, this control logic can be co-located with the database as a background service. We chose to have a per-region centralized service to optimizer the speed of engineering, operationalization, and monitoring. A centralized control plane also allowed a centralized store for the state and details of recommendations, history of actions, etc. This centralization further simplified exposing this information to users who administer several databases, e.g., SaaS vendors and CSVs who manage hundreds to thousands of databases.

We implement the control plane as a collection of micro-services, where each micro-service is responsible for a designated task. The major micro-services (and their tasks) are: (a) invoke database analysis and generation of index recommendations; (b) implement recommendations; (c) validate recommendations; (d) detect issues with the auto-indexing service and taking automated corrective actions or filing a service incident to notify on-call engineers;

Each of these micro-services appropriately update the state machine of a database or a recommendation for a database. The database state corresponds to different auto-indexing configuration settings (see Section 2 for some examples). We will discuss some additional database states as we discuss specifics of the index recommenders (Section 5.3).

A recommendation can be in one of the following states: (i) *Active:* an index create or drop recommendation ready to be applied; (*ii*) *Expired:* a terminal state of a stale recommendation either due to the age of the recommendation (i.e., when it was generated) or a newer recommendation invalidated it; (*iii*) *Implementing:* the recommendation is currently in the process of being implemented on the database (either user-initiated or system-initiated if auto-implement setting is turned on); (*iv*) *Validating:* validation in progress analyzing execution data after the recommendations have been implemented; (*v*) *Success:* a terminal state where a recommendation was successfully applied and validated; (*vi*) *Reverting:* validation detected a query performance regression and the system auto-initiated a revert of the action (e.g., drop a created index); (*vii*) *Reverted:* a terminal state where a recommendation was successfully reverted; (*viii*) *Retry:* a transient error was detected in one of the actions on a recommendation and the failed action will be retried; (*ix*) *Error:* a terminal state where the system encountered an irrecoverable error (e.g., an index with the same name already exists, the table or column was dropped, the index was dropped external to the system, etc.). Many of the above states have sub-states for further diagnosis and automated actions. Some of the error states are well-known and automatically processed, while others are treated as service health issues which will trigger incidents for analysis by on-call engineers.

This state information needs to be stored in a persistent, highly-available data store. Being part of such a highly-available persistent database service, the control plane stores its state in a database within the same region of Azure SQL Database. This database is similar to a customer database in Azure SQL Database with enhanced manageability options since it stores system state instead of customer data.

5 INDEX RECOMMENDER

5.1 Background

5.1.1 Recommendation Source. SQL Server 2005 introduced the *Missing Indexes* (MI) feature where the server, during query optimization, identifies indexes that are not present in the database that can improve this query's performance. These missing indexes are exposed via dynamic management views (DMVs), the special system views in SQL Server, and in the execution plan XML [34]. The benefit of the MI feature is that it is a lightweight always-on feature where missing indexes from across queries are accumulated in the DMVs. However, the lightweight nature also implies that MI only performs a local analysis, predominantly in the leaf node of a plan, to identify a lower bound of improvement. The MI feature cannot recommend indexes that benefit joins, group by, or order by clauses, and does not perform workload-level analysis or the cost of maintaining them [23].

The Database Engine Tuning Advisor (DTA), on the other hand, is a more comprehensive physical design tool initially released with SQL Server 7.0 (or 1998) [10] and significantly enhanced over future releases [2, 17]. DTA takes a workload ($\mathcal{W}$) as input and outputs a recommendation that minimizes the optimizer-estimated total cost of $\mathcal{W}$. DTA can recommend clustered and non-clustered indexes, B+ tree and Columnstore indexes, filtered indexes, materialized views, and partitioning. DTA's candidate selection considers sargable predicates, joins, group by, and order by columns to identify candidate physical designs [22]. In addition, DTA supports constraints such as maximum number of indexes, storage budget, etc. However, in contrast to MI, a DTA tuning session needs to be manually invoked with appropriate settings for these options. Moreover, DTA creates sampled statistics and makes additional "what-if" optimizer calls [11] which result in higher overhead compared to MI.

MI and DTA have complementary benefits. The low overhead of MI allows us to still recommend beneficial indexes for databases with small amounts of resources (e.g., the databases in the Basic tier). On the other hand, DTA performs a more comprehensive analysis which is suitable for more complex applications deployed in the higher-end Premium tier databases. There are several factors that determine which of these two recommendation sources will be used for a given database, e.g., the service tier [28], activity level and resource consumption of the database, etc. A pre-configured policy in the control plane determines which recommender to invoke.

5.1.2 Workload Coverage. Identifying an appropriate set of indexes for a given database requires a holistic analysis of the queries (and updates) on the database. When a DBA drives this tuning, e.g., in an on-premises setting, *representative workload* ($\mathcal{W}$) identified by the expert human helps in this holistic analysis. Tuning tools require this input $\mathcal{W}$ to be provided. Interviewing several DBAs and customers revealed that in many real settings, identifying such a representative workload is challenging even for expert humans due to the complexity and heterogeneity of applications and the complex mapping of application logic to the corresponding SQL statements executed. When automatically indexing without the knowledge of the application or human inputs, it is even hard to define *representative*.

To evaluate the goodness of $\mathcal{W}$ analyzed to generate the index recommendations, we instead rely on *workload coverage*. If the index recommender analyzed a set of statements $\mathcal{W}$, workload coverage is the resources consumed by these statements in $\mathcal{W}$ as a percentage of the total resources consumed by all statements executing on the database. A high workload coverage, e.g.,> 80%, implies that the statements analyzed by the tuner account for a significant amount of resources consumed for the database. We use Query Store's historical execution statistics to compute workload coverage.

5.2 Missing Indexes

When SQL Server query optimizer searches for an execution plan for a query, it analyzes the query predicates to identify the best indexes that provide cheap access to the data relevant to the predicates. If these indexes do not exist, the query optimizer exposes this information through the MI feature [34]. These indexes are accumulated in a set of DMVs. They have associated details of the predicate in the DMV, e.g., which columns are accessed in an equality predicate (EQUALITY columns), which in an inequality predicate (INEQUALITY columns), and which columns are needed upstream in the query plan (INCLUDE columns).

One limitation of the MI feature is that it only provides raw information about index candidates. In order to use these candidates to generate the final recommendations, we need to analyze their holistic impact. The statistics accumulate over time, but are reset after a server restart, failover, or schema change. To tolerate such resets, we take periodic snapshots of the MI DMVs, while keeping the overhead of taking snapshots low. The index recommender performs a workload-level analysis leveraging the raw MI snapshots.

We perform the following high-level steps to generate the final index recommendations. *First*, we define a non-clustered (secondary) index candidate with information in the MI DMV. The EQUALITY columns are selected as keys, INCLUDE columns are the included columns. SQL Server storage engine can seek a B+ tree index with multiple equality predicates but only one inequality predicate. Hence, we pick one of the INEQUALITY columns as a key (ordered after EQUALITY columns), the remaining are included columns. This column choice is deferred to the merging phase.*Second*, the MI DMV provides statistics such as the number of queries that triggered the MI recommendation, the average estimated cost of each query that could be improved, and a percentage improvement expected from the index (in optimizer's estimates). We use these statistics to determine the aggregated *benefit* of an index, using the raw optimizer-estimated benefit for each candidate. *Third*, we filter out candidates that have very few query executions (e.g., ad-hoc queries).

Fourth, since the MI DMV accumulates these statistics, we expect that really beneficial indexes will have an impact that will accumulate and increase over time. However, due to our need to tolerate resets of the DMV, we need a statistically-robust measure of this positive gradient of the impact scores over time. We formulate this as a hypothesis test. Assuming that these errors are distributed normally, we compute the t-statistic on the slope of an index's impact over time being above a configurable threshold. By analyzing these statistics over hundreds of thousands of databases, we found that for high-impact indexes, a few data points are sufficient to surpass the pre-determined certainty limit. *Fifth*, to identify opportunities for indexes that benefit multiple queries, we merge indexes [12]. We perform conservative merging of indexes, e.g., merge candidates whose key columns are a prefix of another, but include columns differ. We merge indexes only if the improve the aggregate benefit across queries that the merged impact improves. As the *last* step, we identify the top indexes with the highest impact with an impact slope above the threshold. Since we do not make additional optimizer calls for the workload-level analysis, we use a classifier to further filter out index recommendations which are expected to have low impact on execution. We use data from previous index validations and features such as estimated impact, table and index size, etc. to train a classifier that identifies such low impact indexes.

This approach does not use an explicitly-specified $\mathcal{W}$. Missing indexes are analyzed for every statement, except inserts, updates, and deletes without predicates. Hence, workload coverage is all resources except the percentage consumed by the above statement types.

5.3 Database Engine Tuning Advisor

DTA is a comprehensive physical design tool that given a workload $\mathcal{W}$, finds the physical design that minimizes the optimizer-estimated cost of $\mathcal{W}$ [2, 10]. DTA uses a cost-based search where for every query $Q \in \mathcal{W}$, DTA starts with candidate selection that finds the optimal configuration for Q. Candidates from all queries are used for a workload-level enumeration, which outputs the final recommendation. DTA uses the query optimizer's "what-if" API to cost hypothetical configurations during its search [11]. DTA is designed to be invoked by a DBA. To leverage DTA for auto-indexing in Azure SQL Database, we rearchitected it to run as an automated service. Our enhancements and design changes fall in three broad categories: (a) ability to run DTA with strict resource budget and minimal impact on any production workloads; (b) automatically acquiring a representative workload $\mathcal{W}$ for tuning; (c) running DTA as a service at the scale of Azure SQL Database.

5.3.1 *Reducing DTA overheads.*

DTA needs to connect to the database to access metadata, build and read sampled statistics, and make query optimizer calls. Due to the number of calls DTA makes to the server and to satisfy security and compliance requirements, we need to run DTA co-located with the primary copy of the database server in Azure SQL Database, running concurrent with the customer's workload. Therefore, DTA needs to run with a stringent resource budget and ensure minimal impact on the customer's workload.

There are two ways DTA can impact customer's workloads: (i) *Resources consumed by DTA:* for resources consumed on the server with optimizer calls, creating sampled statistics, etc., we rely on SQL Server's resource governance mechanisms to limit CPU, memory, and I/O consumed by DTA calls [15, 25]. For resources consumed by the DTA process, we use Windows Job Objects [45]. (*ii*) *Lock and latch contention:* caused due to creation and dropping of hypothetical indexes and sampled statistics. Such lock contention, especially for modifying metadata, can cause significant impact on user workload due to the FIFO nature of SQL Server's lock scheduler. We rely on low priority locking support that allows DTA to request a lock with lower priority, without blocking lock requests by user's workload [43]. We also made several changes to DTA to reduce the overheads, e.g., we reduced the number of sampled statistics created by DTA by $2-3\times$ without noticeable impact on recommendation quality. To further minimize the resource footprint, the control plane only invokes DTA on-demand when we need to analyze a database. Since these approaches only reduce the chances of impacting the user workload, we added automated tracking to detect instances where a DTA session is slowing down user queries and abort such a DTA session. We leverage SQL Server's detailed wait statistics, blocked process reports, and signals from Azure SQL Database's Intelligent Insights feature [19] to detect such slow downs.

5.3.2 Identifying a Workload. We need to automatically derive $\mathcal{W}$ for DTA to tune. We leverage past execution statistics and query text captured by Query Store [29] to construct $\mathcal{W}$. At the start of a DTA session, we analyze the execution statistics for the past $\mathcal{N}$ hours to identify the $\mathcal{K}$ query statements (or templates) which are most expensive in terms of duration or resources (e.g., CPU time). Since this analysis can be resource intensive for database with many queries accumulating gigabytes of data in Query Store, we set $\mathcal{N}$ and $\mathcal{K}$ based on the amount of resources available to the database. While other approaches can be used to select $\mathcal{W}$, the above approach efficiently identifies the most important statements, balancing workload coverage with the resources spent on workload analysis.

Identifying the statements to tune is only a start. DTA needs the full query statements in $\mathcal{W}$ to estimate its cost for the different hypothetical index configurations [11]. While Query Store captures the query text, it is designed to help users get context about a query when debugging performance issues, and not as a detailed workload capture tool (which is often too expensive to be always-on like Query Store). T-SQL, SQL Server's dialect of SQL, supports complex constructs such as conditionals, loops, scalar and table-valued variables. As a result, statements in Query Store are often incomplete, e.g., missing variables, or only having a fragment of a complex conditional statement, etc. Such statement text cannot be costed by DTA via the "what-if" API. Another challenge arises from statements that SQL Server cannot optimize in isolation or is not supported in the "what-if" API. For instance, in a batch of T-SQL statements, one statement can store a result of a query into a temporary table, and then another statement can reference this temporary table. Such batches in SQL Server can only be optimized during execution. These limitations restrict the set of statements DTA can successfully tune, often significantly impacting DTA's workload coverage.

We made several enhancements to DTA to overcome these limitations. *First*, while Query Store is our primary source of workload, we augment query text from other sources. For instance, if statements are part of stored procedures or functions whose definition is available in system metadata, we obtain the statements from metadata. *Second*, for incomplete T-SQL batches, we use SQL Server's query plan cache to obtain the full batch definition if available. *Third*, we rewrite some types of statements from the original form, which cannot be optimized in the "what-if" mode, into equivalent statements which can be successfully optimized. For instance, `BULK INSERT` statement used by bulk load tools or APIs cannot be optimized. DTA rewrites them into equivalent `INSERT` statements which can be optimized, thus allowing it to cost the index maintenance overheads due to these operations. *Last*, for all of the above statement types, the MI feature generates candidates if these statements can benefit from indexes. Hence, we augment DTA's search with these MI candidates. We use the optimizer's cost estimates when generating the MI candidates whenever DTA cannot cost them using the "what-if" API, thus allowing such candidates to be used in DTA's cost-based search.

Once DTA completes analyzing a database, it emits detailed reports specifying which statements it analyzed and which indexes in the recommendation will impact which statement. We use this report to expose the recommendation details to the users. We also use these reports to compute the *workload coverage* which provides us an approximate measure of the effectiveness of DTA's recommendations. Identifying patterns in errors for statements DTA is unable to process prioritizes future DTA improvements.

5.3.3 Running DTA as a Service. A DTA tuning session for a database can run for minutes to hours, depending on the complexity of the workload, schema, and available resources. There can be several types of failures during a DTA session, either on the database server, the physical server, or DTA. There could be DTA sessions on thousands of databases at a given instant of time. To tolerate these failures and manage the state of DTA sessions at scale, we created a micro-service in the control plane dedicated to DTA session management, and augmented a database's state with DTA session states. This micro-service identifies when to invoke a DTA session on a given database, tracks progress of the DTA session, and ensures that the session reaches a terminal state of either successful completion or an error which would trigger a cleanup (to remove temporary objects such as hypothetical indexes and statistics) and an optional retry.

A single DTA session involves a complex cost-based search through a huge space of alternatives. Debugging the recommendation quality for a given session is in itself a daunting task when DTA is tuning a large and complex workload. We need to be able to identify issues with thousands of such sessions without knowledge of the database DTA is tuning. Using carefully selected telemetry from DTA and correlating with other Azure SQL Database telemetry, we determine the overall health of DTA sessions, detect anomalies, potential issues, and identify avenues to further improve DTA.

5.4 Dropping Indexes

As the workload, schema, and data distributions evolve and new indexes are created, the benefit of existing indexes can decrease. It is therefore important to identify and potentially drop such low impact indexes to reduce their maintenance overhead and storage space. Our analysis also revealed that many databases often have several duplicate indexes, i.e., indexes with identical key columns (including identical order), which are also candidates to be dropped.

Dropping indexes pose several challenges. *First*, users often have indexes for occasional but important queries, such as reports, at some cadence such as daily or weekly. Since we automatically identify $\mathcal{W}$, such infrequent events may not be part of $\mathcal{W}$ and hence ignored in analysis. Dropping such indexes can cause significant slowdown for these occasional queries. Such regressions are also hard to detect in validation since the queries are infrequent. *Second*, it is common for queries to hint indexes when users manually tune queries [35] or force a plan [27]. Dropping such a hinted index would prevent the query from executing, potentially breaking the application. *Third*, identifying which among the duplicate indexes to drop can also be a challenge. In many cases, retaining any one of them is acceptable, while in some other cases, a specific one (or many) may be preferred.

We take a conservative approach to identify indexes that if dropped will have minimal risk of disruptions. Instead of being workload-driven, we leverage other sources of information from SQL Server to identify indexes with little or no benefit to queries. *First*, we analyze statistics tracked statistics tracked by Azure SQL Database, such as how frequently an index is accessed by a query, how much it is modified etc., to identify indexes that do not benefit queries but have significant maintenance overheads. *Second*, to prevent disrupting applications, we eliminate candidates that appear in query hints or forced plans, or are enforcing an application-specified constraint. We analyze statistics over a long time period (e.g., 60 days). To reduce storage overheads for this long term data retention, we leverage Azure SQL Database's telemetry and offline analysis systems to analyze this data and identify drop candidates.

6 IMPLEMENTATION AND VALIDATION

Implementation. When a user decides to apply a recommendation or if auto-implementation is enabled, the control plane orchestrates index implementation (either create or drop) and subsequent validation. Depending on the index size, creation can be a resource-intensive operation that scans the data (I/O intensive), sorts it (CPU and memory intensive), and then creates the index (log intensive). We minimize the impact of this resource-intensive task on concurrent user workload by (i) governing the resources [15, 25]; and (ii) scheduling most of the operations during periods of low activity for the database. To further minimize impact, our service only supports online operations, i.e., operations that can be completed with minimal or no blocking. Since this creation operation can be long-running, a micro-service in the control plane tracks the state machine of this implementation step to ensure we tolerate the different errors and failures during index implementation.

Validation. The goal of validation is to detect and correct major query performance regressions caused by index creation or drop. We leverage Query Store [29] to analyze execution statistics before and after we implement the index change. One major challenge we encounter in validation is the inherent noise in the execution statistics due to concurrent query executions in an uncontrolled production setting.

First, we focus on logical execution metrics such as CPU time consumed, or logical bytes read. These metrics are representative of plan quality and also have less variance compared to physical metrics such as query duration or physical I/O. If the logical metrics improve due to an index, metrics such as duration generally improve. *Second*, we only consider queries that have executed before and after the index change and had a plan change due to the index change. That is, if an index is created, the new plan after creation should reference the index, while if an index is dropped, the old plan before drop should reference the index. *Third*, for every query plan, Query Store tracks the number of executions, average, and standard deviation for every metric (e.g., CPU time). Assuming the measurement variance follows a normal distribution, we use the above statistics and Welch t-test [42] to determine the statistical significance of improvement or regression of the above-mentioned metrics. We compare the state after the index change with that before the change.

If a significant regression is detected, the system automatically reverts the change, i.e., drops the index if created, or recreates it if dropped. The trigger to revert can be set

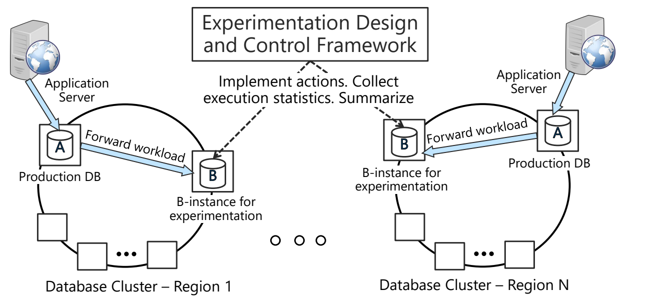

Experimentation Design and Control Framework
Implement actions. Collect execution statistics. Summarize
Application Server
Forward workload
A
B
Production DB
B-instance for experimentation
Database Cluster – Region 1
Application Server
Forward workload
Production DB
B-instance for experimentation
Database Cluster – Region N

Figure 5: A conceptual architecture for experimentation using B-instances in Azure SQL Database.

to a conservative setting where a regression for any statement that consumes a significant fraction of the database's resources can trigger a revert. Without explicit application knowledge or user inputs, this approach minimizes disruption, though might also reduce the workload-level improvement possible. An alternative setting measures the holistic improvement of all statements affected by the index, and reverts only if there is a regression at an aggregate level. This approach may significantly regress one or more statements if improvements to other statements offset the regressions.

7 EXPERIMENTATION IN PRODUCTION

Azure SQL Database presents us a unique opportunity to experiment at scale with a diverse collection of production workloads, which we can use to build confidence on our recommendation quality or test major changes. This experimentation is similar to A/B testing used in randomized experiments [1], but adapted to the context of databases.

7.1 B-instances

Databases exhibit huge diversity in schema, queries, data sizes, and data distributions. Hence, if we want to experiment with two index recommenders and measure their quality in terms of execution cost, we need the same set of databases to generate recommendations, implement them, and compare the execution costs before and after the implementation. Performing this on the primary database copy serving application traffic is risky and unacceptable. Even using a secondary replica is not feasible since Azure SQL Database requires the replicas to be physically identical.

Our ability to create a B-instance of a database in Azure SQL Database allows us to experiment at scale in production without the risk of affecting customer workloads, while maintaining Azure's compliance, security, and privacy requirements. A B-instance is a different database instance (invisible to external customers) that starts with a snapshot of the database. It is on a different physical server within the compliance boundary of Azure SQL Database. It has an independent set of resources and different security credentials and can also use a different database server binary. A B-instance receives a fork of the Tabular Data Stream (TDS) [38] traffic from the primary copy (called an A-instance in this context) which is replayed on the B-instance. Resources for the B-instance are accounted as Azure SQL Database's operational overhead and not billed to the customer. We use several levels of safeguards and isolation to ensure that the B-instance cannot jeopardize the A-instance if it misbehaves. To eliminate any expensive synchronization, the B-instance independently executes the TDS stream, allowing reordering of operations. A failure of the B-instance does not affect the A-instance's ability to progress normally. Hence, the B-instance is a best-effort clone and can potentially diverge from the A-instance due to dropped or reordered operations.

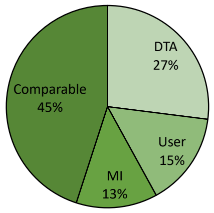
7.2 Experimentation Framework

Creating a B-instance is one small step in an experiment. We need to randomly choose databases that meet the criteria for an experiment, generate index recommendations, implement them, collect execution statistics, and analyze execution costs with and without the recommendations. Different experiments may also need custom steps, e.g., dropping a subset of existing indexes, or enabling certain features. To scale these experiment steps to hundreds or thousands of databases across different geographical regions, we built a framework for experiment design and control. Figure 5 presents a conceptual architecture of the experimentation framework leveraging B-instances. It is a workflow engine where different tasks of the experiment can be defined as workflow steps which are then stitched to define an experiment workflow. The framework executes the workflow on each database identified as a candidate for the experiment, monitors the state of the workflow, and takes corrective or cleanup actions if an error is detected. The framework has a library of commonly-used steps, e.g., creating a B-instance, detecting divergence of a B-instance, common analysis steps etc. It allows custom steps to be added for any experiment.

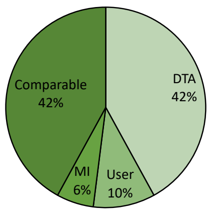
7.3 Experimentation Results

One major question for us was the quality of recommendations, in actual execution cost and not estimates, that the different recommenders generate for the diverse collection of workloads in Azure SQL Database. To answer this question, we summarize experiments performed on a few thousand production databases, between March and June 2017, leveraging our experimentation framework and B-instances.

We experiment with the two index recommenders we developed–using Missing Indexes (*MI*) and using Database



(a) Premium tier.



(b) Standard tier.

Figure 6: Experimentation at scale with production databases in the premium and standard tier.

Engine Tuning Advisor (*DTA*)–to better understand the trade-offs between them and to determine a policy to best leverage their abilities to meet the diversity of Azure SQL Database. We also compare these advisors with how human administrators tune their databases (*User*). We develop a heuristic to emulate this *user's* tuning at the scale of our experiments. We identify highly beneficial non-clustered (secondary) indexes *already existing* in the database but do not enforce any application constraint. Our hypothesis is these indexes were added by the user to improve performance. Using statistics tracked in Azure SQL Database such as the `dm_db_index_usage_stats` DMV and Query Store, for each database, we identify the N existing indexes the provide the most benefit to queries. We then select a random subset of k indexes to drop. For these experiments, we used $N = 20, k = 5$. We assume the performance after dropping the k indexes was performance before *User* tuned the database, and performance with the k indexes is the improvement due to *User's* tuning. After dropping the indexes, we let *MI* and *DTA* recommend up to k indexes to create.

We design the experiment in phases where each phase measures the impact of one of the recommenders and collects execution statistics for more than a day. For the phase implementing MI or DTA recommendations, these indexes are reverted at the end of the phase. We do not control the arrival or concurrency of the queries executing on the databases, since the B-instance is replaying the workload from the TDS fork of the A-instance. To tackle this variance in measurements due to concurrency, diurnal, or other temporal effects, we use statistical tests, such as Welch t-test [42], to determine the statistical significance of changes in execution costs. We measure the average and standard deviation of several metrics (e.g., CPU time, logical reads). Our analysis uses a fixed execution count for all phases to address the different number of executions of a query over the phases.

We run two sets of experiments by randomly selecting active databases from two service-tiers in Azure SQL Database: (*a*) *premium* which comprises high-paying customers with business-critical workloads and more complex applications; and (*b*) *standard* which is a cheaper tier where databases have much fewer resources. Figure 6 presents a pie chart of the percentage of the databases where a specific recommender's indexes outperformed the others by improving CPU time. To account for variance, we only consider statistically significant improvements. For instance, the slice corresponding to *DTA* implies indexes recommended by DTA outperformed both user's and MI's recommendations. The slice *Comparable* corresponds to databases where the performance of all three alternatives was indistinguishable.

As evident from Figure 6, no one recommender dominates on all databases. It was encouraging to see that our automated techniques were able to match or outperform human administrators in $85 - 90\%$ of databases. However, for more complex workloads and expert users in the premium tier, these approaches do not outperform expert DBAs. We also compute the CPU time improvement due to each recommender of the entire workload on a database, and average it across all the databases. We observed *DTA* results in $\sim 82\%$, *MI* results in $\sim 72\%$, and *User* results in $\sim 35\%$ CPU time improvement. That is, auto-indexing can unlock even more significant improvements compared to that of *User*.

8 OPERATIONAL EXPERIENCE

8.1 Operational Statistics

Here we present a snapshot of some statistics from the operational service as of October 2018. Index recommendations are generated for all databases in Azure SQL Database, with $\sim 250K$ recommendations to create new indexes and $\sim 3.4M$ recommendations to drop existing indexes that are not benefitting queries. About a quarter of the databases have auto-implementation enabled where about $\sim 50K$ indexes are created and $\sim 20K$ indexes dropped on an average week. Since our service is tuning databases for more than two years, we observe many databases reach a steady state with only occasional new index recommendations generated for them. However, we also observe an increasing stream of new databases that are deployed. There are hundreds of thousands of queries whose CPU time or logical reads reduced by $> 2\times$ due to indexes recommended by the service, and tens of thousands of databases where the workload's aggregate CPU consumption reduced by $> 50\%$.

In aggregate, $\sim 11\%$ of our automated actions are reverted due to validation detecting regressions. Since the *MI*-based recommender does not account for index maintenance overheads, many reverts are due to writes becoming more expensive. For both recommenders, a significant fraction of reverts are due to regressions in `SELECT` statements where optimizer's errors results in query plans estimated to be cheaper but is more expensive when executed. Reducing regressions and revert rates is an area of future work.

8.2 Customer Feedback

Setting and meeting customer expectations added an interesting dimension to our operational experience. Customer feedback spanned the spectrum of being delighted to cautious skepticism. Many customers, especially large customers managing hundreds to thousands of databases, saw immense value from the savings to their database management costs. They embraced the service and provided useful feedback. Several interesting case studies are presented in [41]. Example customers such as SnelStart, an invoicing and bookkeeping application, AIMS360, a cloud technology provider for the fashion industry, and several large internal applications at Microsoft, all of which have hundreds to tens of thousands of databases in Azure SQL Database, saw positive results by enabling auto-implementation of recommendations, often stating that the service *"has allowed them to completely stop proactively thinking about index management."*

Earning the customer's trust is one of the major challenges for an automated indexing system for production scenarios. Lack of trust is also a major hindrance in adoption for automated implementation. Customers seeking to build trust on the feature had the following concerns: (*i*) *business continuity*: automatically applied indexes do not cause performance regressions or other problems such as blocking schema changes; (*ii*) *meaningful performance gains:* the resources spent in analyzing and building indexes result in noticeable performance improvements over time; (*iii*) *transparency:* provide a history of the actions and their impact on performance; (*iv*) *robustness and reliability:* handles the different failure cases and reliably tunes the database over time, even if the recommendations are not optimal.

Addition of the validation module to detect regressions and automatically revert the indexes, resource governance and other measures to ensure minimal impact on the workload, exposing the history of actions along with queries impacted, maintaining a high quality of recommendations, and exposing the option of disabling auto-implementation to put the user in control contributed significantly to increase the customer's trust on the system. Most customers were comfortable with the fact that in a small fraction of cases, an index might regress performance, as long as a corrective action automatically and reliably fixes the issue.

Many customers seek to exercise more control: (*i*) how and when the indexes are implemented, e.g., implementing indexes during low periods of activity or on a pre-specified schedule; (*ii*) how to share resources between index creation and concurrent transactions; (*iii*) naming scheme for indexes. Customers managing hundreds of similar databases, e.g., a SaaS vendor, desire features such as only implement indexes that are beneficial for a significant fraction of their databases.

Another customer ask was the integration of automated index management with other application development and schema management tools, such as Visual Studio, SQL Server Data Tools, or even third party management tools. This will enable tracking versions of the schema and indexes and better integration with deployment tools. Developers maintain the logical database model in such tools. Lack of these integrations implied automatically implemented indexes were not propagated to the database model. Hence, every new deployment of the application would drop these indexes, only to be recreated by the auto-indexing service.

8.3 Operational Challenges and Lessons

Operationalizing the service at Azure-scale posed several challenges. Some are common to many cloud services that operate at this scale with several dependencies. Many others are specific to indexing and the complexity of applications leveraging many advanced Azure SQL Database features.

Monitoring this decentralized service spread across different geographic regions posed a major challenge. We rely on dashboards to aggregate data from disparate regions to create an aggregated view of the service for engineers to uncover issues and find avenues to improve the service further. The auto-indexing service has dependencies on other services in Azure SQL Database or Azure. Any such service can fail independently and our service needs to gracefully handle these failures. Scaling auto-indexing to all databases in Azure SQL Database generates vast amounts of state that the control plane (see Section 4), which varies from one region to another. Appropriately provisioning resources for the control plane requires careful planning.

Creating an index can take minutes to hours, especially for databases in lower service tiers with small amounts of resources. Index creation generates a significant amount of log, which cannot be truncated until index is created. We encountered several instances where we filled up a database's transaction log when creating an index on a large table. Recent improvements in Azure SQL Database, such a resumable index create [37], help us address this challenge by allowing more frequent log truncation even if the index is not fully created. We can also pause/resume a creation if there is high contention for resource or a failure during index build.

Metadata contention when creating and dropping indexes posed an interesting challenge. We carefully chose operations which can be performed online without blocking concurrent queries, e.g., indexes that can be created online. However, reverting an index that caused regressions posed a challenge. To drop an index, the server must ensure that no concurrent query optimization or execution is using the index being dropped. Hence, this operation requires an exclusive schema lock on the table. Similarly, any statement referencing the table will acquire a shared schema lock. Since SQL Server's lock scheduler is FIFO, if a request for an exclusive lock is blocked behind one or more shared locks by long-running queries, it will in turn block a subsequent shared lock request. That is, even though dropping a non-clustered index is a lightweight metadata operation, it can block other concurrent transactions and create convoys. In rare instances, it could significantly disrupt the application's workload. To overcome this issue, we use managed lock priorities in SQL Server to first issue the drop index request at a low priority that does not block concurrent user transactions [43]. We use a back-off and retry protocol if this request times out. The control plane manages this fault-tolerant protocol.

A major requirement to gain customer trust was to ensure auto-created indexes do not block application processes such as schema changes, application upgrades, and bulk data loads. Hence, we added mechanisms to cascade and force the drop of auto-created indexes which block customer-initiated actions such as dropping a column which is also part of an auto-created index, or partition switching [44].

Azure SQL Database supports a rich set of features which are used by different applications. Features such as in-memory tables, Columnstore, full-text indexing, different data models such as XML and Graphs, etc. have very different indexing characteristics and restrictions. Almost all simplifying assumptions eventually break. Therefore, we intentionally optimize for the most frequent scenarios but gracefully handle the edge cases to ensure our indexes do not conflict with these advanced features.

9 RELATED WORK

Automatically finding the right set of indexes has been an active area of research for several decades [18, 36]. The late nineties saw several commercial index tuning tools from the major database vendors [10, 14, 40]. These tools were extended to support other physical design structures such as materialized views and partitioning [2, 3, 30, 46]. Database Engine Tuning Advisor (DTA) [2] was recently extended to support both Columnstore and B+ tree index recommendations [17]. Our service leverages the decades of advances in DTA and MI features in SQL Server, and complements them with technology necessary to run it as a service that automatically analyzes the workload, recommends indexes, optionally implements them, detects and corrects any query performance regressions caused by index changes.

The state-of-the-art commercial physical design tools for RDBMSs have focused on assisting a human expert, such as a DBA, in the complex search space of alternatives. They put the human in the loop of tuning, require several critical inputs (e.g., the workload, when to tune). They also rely on the human to analyze the recommendations to determine which recommendations to implement. Variants of the problem have been studied in research such as continuous tuning [9, 31], using the DBA in the loop [32], or leveraging machine learning techniques to completely automate indexing [26]. This paper is the first industrial-strength solution for fully-automated indexing.

10 CONCLUDING REMARKS

We presented the design, implementation, experience, and lessons learned from building the first industrial-strength auto-indexing service for Microsoft Azure SQL Database. An industry-leading product offering that automatically recommends, implements, and validates indexes, such automation reduces our customer's burden of performance tuning. While our architecture was developed in the context of the Azure SQL Database cloud platform, we expect many ideas can be leveraged in an on-premises setting or in other cloud database services. We also presented a novel approach for experimentation in a cloud database service by leveraging B-instances with no risks to production workloads. Such experimentation helped us get more confidence on the recommendation quality.

We presented several key learnings from operationalization and customer feedback. One meta learning was that we can provide significant value to a large fraction of our customers. However, it is incredibly difficult to cover all the special cases that arise with all the features of a modern RDBMS. Looking ahead, gaining trust of more customers will require covering the special cases our expert customers care about. These include giving more control where customers desire (e.g., broader criteria for selecting $\mathcal{W}$), further improve our recommendation quality and reduce performance regressions caused by our recommendations, improving workload coverage and identifying more complex workload patterns, and making this auto-indexing service transparent to our customers by further reducing its impact.

ACKNOWLEDGMENTS

We thank the entire *auto tuning* engineering team in Azure SQL Database for their contributions to the auto-indexing service. In particular we thank, Iva Lazovic, Danimir Ljepava, Aleksandar Milanovic, Vladan Simov, and Veljko Vasic. We are also grateful to our customers for trusting us and providing useful feedback. We thank Bailu Ding and Wentao Wu for their suggestions on the paper, and Tomas Talius for his help with B-instances and experimentation.

REFERENCES

[1] A/B testing 2018. A/B testing. https://en.wikipedia.org/wiki/A/B_testing.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. 1110–1121.

[3] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD*. 359–370. https://doi.org/10.1145/1007568.1007609

[4] aprc [n. d.]. Automatic Plan Correction in Microsoft SQL Server 2017 and Azure SQL Database. https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning.

[5] autoindexingazuredb [n. d.]. Automatic Index Tuning in Azure SQL Database. https://blogs.msdn.microsoft.com/sqlserverstorageengine/2017/05/16/automatic-index-management-in-azure-sql-db/.

[6] autotuningazuredb [n. d.]. Automatic Tuning in Azure SQL Database. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-automatic-tuning.

[7] Azure Global Infrastructure 2018. Azure Regions. https://azure.microsoft.com/en-us/global-infrastructure/regions/.

[8] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. 2012. Automated physical designers: what you see is (not) what you get. In *DBTest*. 9. https://doi.org/10.1145/2304510.2304522

[9] Nicolas Bruno and Surajit Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *ICDE*. 826–835. https://doi.org/10.1109/ICDE.2007.367928

[10] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 146–155. http://dl.acm.org/citation.cfm?id=645923.673646

[11] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378. https://doi.org/10.1145/276304.276337

[12] Surajit Chaudhuri and Vivek R. Narasayya. 1999. Index Merging. In *ICDE*. 296–303. https://doi.org/10.1109/ICDE.1999.754945

[13] Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*. 3–14. http://www.vldb.org/conf/2007/papers/special/p3-chaudhuri.pdf

[14] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *VLDB*. 1098–1109. http://www.vldb.org/conf/2004/IND4P2.PDF

[15] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. 2013. CPU Sharing Techniques for Performance Isolation in Multitenant Relational Database-as-a-Service. *PVLDB* 7, 1 (2013), 37–48. https://doi.org/10.14778/2732219.2732223

[16] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2018. Plan Stitch: Harnessing the Best of Many Plans. *PVLDB* 11, 10 (2018), 1123–1136. http://www.vldb.org/pvldb/vol11/p1123-ding.pdf

[17] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree - Are Hybrid Physical Designs Important?. In *SIGMOD*. 177–190. https://doi.org/10.1145/3183713.3190660

[18] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.* 13, 1 (1988), 91–128. https://doi.org/10.1145/42201.42205

[19] Intelligent Insights 2018. Intelligent Insights using AI to monitor and troubleshoot database performance. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-intelligent-insights.

[20] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[21] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem? http://wp.sigmod.org/?p=1075.

[22] Missing Indexes and DTA 2018. Related Query Tuning Featurese. https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms345577(v=sql.105).

[23] Missing Indexes feature 2018. Limitations of the Missing Indexes Feature. https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms345485(v=sql.105).

[24] MYOB Case Study 2018. Microsoft Azure Case Studies - MYOB. https://azure.microsoft.com/en-us/case-studies/customer-stories-myob/.

[25] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR*. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

[26] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*. http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf

[27] planguide [n. d.]. Plan Guides. https://docs.microsoft.com/en-us/sql/relational-databases/performance/plan-guides.

[28] Purchasing models 2018. Azure SQL Database purchasing models. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-service-tiers.

[29] Query Store 2018. Microsoft SQL Server Query Store. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store.

[30] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. 2002. Automating physical database design in a parallel database. In *SIGMOD*. 558–569. https://doi.org/10.1145/564691.564757

[31] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: continuous on-line tuning. In *SIGMOD*. 793–795. https://doi.org/10.1145/1142473.1142592

[32] Karl Schnaitter and Neoklis Polyzotis. 2012. Semi-Automatic Index Tuning: Keeping DBAs in the Loop. *PVLDB* 5, 5 (2012), 478–489. https://doi.org/10.14778/2140436.2140444

[33] SnelStart Customer Story 2018. With Azure, SnelStart has rapidly expanded its business services at a rate of 1,000 new Azure SQL Databases per month. https://customers.microsoft.com/en-us/story/with-azure-snelstart-has-rapidly-expanded-its-business-services.

[34] SQL Server Documentation 2018. About the Missing Indexes Feature. https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms345524(v=sql.105).

[35] SQL Server Query Hints [n. d.]. Hints (Transact-SQL) - Query. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query.

[36] Michael Stonebraker. 1974. The choice of partial inversions and combined indices. *International Journal of Parallel Programming* 3, 2 (1974), 167–188. https://doi.org/10.1007/BF00976642

[37] Mirek Sztajno. 2018. Resumable Online Index Create is in public preview for Azure SQL DB. https://azure.microsoft.com/en-us/blog/resumable-online-index-create-is-in-public-preview-for-azure-sql-db/.

[38] TDS 2018. Tabular Data Stream. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store.

[39] Umbraco 2018. Umbraco uses Azure SQL Database to quickly provision and scale services for thousands of tenants in the cloud. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-implementation-umbraco.

[40] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110. https://doi.org/10.1109/ICDE.2000.839397

[41] Veljko Vasic. 2017. Artificial Intelligence tunes Azure SQL Databases. https://azure.microsoft.com/en-us/blog/artificial-intelligence-tunes-azure-sql-databases/.

[42] B. L. Welch. 1947. The generalization of 'Student's' problem when several different population variances are involved. *Biometrika* 34, 1-2 (1947), 28–35. https://doi.org/10.1093/biomet/34.1-2.28

[43] What's new in SQL Server 2014 (Database Engine) 2018. Managing the Lock Priority of Online Operations. https://docs.microsoft.com/en-us/sql/database-engine/whats-new-in-sql-server-2016?view=sql-server-2014#Lock.

[44] Cathrine Wilhelmsen. 2015. Table Partitioning in SQL Server âĂŞ Partition Switching. https://www.cathrinewilhelmsen.net/2015/04/19/table-partitioning-in-sql-server-partition-switching/.

[45] Windows Job Objects 2018. Job Objects. https://docs.microsoft.com/en-us/windows/desktop/ProcThread/job-objects.

[46] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*. VLDB Endowment, 1087–1097. http://dl.acm.org/citation.cfm?id=1316689.1316783