Xldb: Execution Log DataBase

Sahil Gandhi
University of California, Los Angeles
Los Angeles, California
sahilmgandhi@cs.ucla.edu

ABSTRACT

The size and the complexity of the Windows, Office, and other related Microsoft codebases are amongst the largest in the world. An internal distributed build tool known as BuildXL was created to speed up the build and testing phases of these codebases and is widely considered a success, bringing down builds from many days (for the largest Windows builds across many versions/languages) to just about 12 hours. These immense builds create immense logs, and so to conserve space, a custom binary format was created and used for the last couple years. This format was prone to breaking log analyzers every time new fields were added or deleted from log entries, and was becoming a growing pain. My internship project was to create a fully forwards-backwards compatible log format for this tool. In the process of the designing and implementing the format, Xldb (Execution Log Data Base) was created. Xldb is a distributed logging system built on top of RocksDb and ProtoBuf3 that enables analyzers to be compatible with different log formats (with a well controlled and designed evolving schema) and offers anywhere between 3x and 150x better analyzing performance than the previous log format (without sacrificing any information). There is a 1.5x-2.5x larger storage footprint associated with the new log format, but the performance gains and compatibility greatly outweigh the costs. Part of the design and implementation was finalized during my internship at Microsft, and the rest was worked on it after as my capstone project.

CCS CONCEPTS

• **Computer systems organization → Distributed systems**; **Logging Infrastructure**; **Databases**.

KEYWORDS

Distributed build engine, RocksDB, ProtoBuf, Logging, Evolving Schemas

1 INTRODUCTION

Microsoft owns and operates some of the largest codebases in the world with the Windows Operating System, Office Product Suite, Bing, and more. While any individual developer only works on a small section of the codebase, the organization as a whole must still run full builds. Often times these are also run across multiple versions as these may all still be in support - Microsoft is (as many forget) still a B2B business in many aspects, and these products tend to have long lives - and the amount of code that must be recompiled and re-tested increases. A full Windows build across several versions and languages could easily run for several days on a single, dedicated build machine and would go through tens of millions of lines of code. This issue is not just prevalent at Microsoft, but indeed at any large corporation with a mature codebase. To tackle this issue, Google created (and later open sourced) a distributed build system called Bazel that is able to take large code compilations and split up the work amongst many different machines. Several other companies have created similar alternatives, but none have gotten as much traction or community support as Bazel.

Microsoft primarily uses the Windows build system (C#, .NET, MSBuild) which at the current moment is not supported by Bazel [6]. The solution here was to create their own in-house alternative known as Domino (later renamed to BuildXL when it became open sourced). BuildXL consists of a front-end where individuals can describe their build process, much like a set of gradle files, and a back-end that schedules and executes individual components on a cluster of machines. BuildXL has been in use at Microsoft for last three to four years and has been gaining traction since it was introduced at a rapid pace. One of the most important aspects of this system is its logs. These logs can range from several megabytes to tens of gigabytes and are proportional to the size of the builds themselves. Engineers and managers often crawl these logs to search for ways to make the builds faster or to look for regressions and other errors that may pop up. Due to the enormous nature of these logs, the team had created a specific binary compressed format, but this format is quite prone to being broken as it is directly linked to the build engine. On average the log analyzers that are used to crawl the logs break once every two to three weeks and this has become quite a nuisance for the relevant engineers. The purpose of Execution Log DataBase (Xldb) is to untie the log and log analyzers from the engine and offer speedups and features requested by many engineers.

1.1 Contributions

Specifically the contributions of my internship and followup capstone-project work are as follows:

- Utilizing RocksDB as the underlying database for the logs, created initial speedups of 5x - 200x for analyzers at a cost of increasing the storage footprint by 2x - 3x.
- Iterated on the DB design and lowered the speedup to 3x - 150x but also lowered footprint size to 1.5x - 2.5x.
- Designed ProtoBuf key/values to ensure backwards and forwards compatibility and reduce the amount of "breakage" of logs and analyzers.
- Decoupled analyzers from the engine allowing more freedom for developers who do not need to recompile engine to crack open logs.
- Introduced further API bindings to ensure a wider variety of developers could analyze the logs in the language they were most comfortable using.

2 BACKGROUND AND MOTIVATION

2.1 BuildXL

Unfortunately, Bazel does not support the Windows build system (C#, .NET, MSBuild), and so Microsoft Research was tasked with creating a compatible solution. The resulting system was BuildXL (formerly known as Domino before being open sourced, and sometimes abbreviated as BXL), which is a state of the art distributed build tool meant to tackle large, complex builds. It consists of two parts, a front-end which is written in a language called DScript, which is a subset of the TypeScript language, and a back-end written in C#. BuildXL was initially created to run on Windows machines only, but with the recent push for the cross-platform .NET Core, a MacOS version was also released, and a Unix version is in the backlog as well. An overview of BXL can be found in Figure 1

2.1.1 BXL Front-End.

The DScript front-end for BXL is designed with simplicity in mind. Users can specify their dependencies, executables, publishable packages (ie. as Nuget packages) and more through these scripts. Users can also choose to explicitly specify output artifacts (files, executables, etc) or implicitly specify just the output directories as "opaque directories" which may have any number or kind of artifacts within. In general, the explicit outputs allow for stricter and more deterministic builds that can be easily debugged and reproduced. This contrasts with much less precise output build systems like MSBuild which is only at the project level, or CMake which is only at the Ninja target level. Furthermore, as DScript is a subset of TypeScript, it is much easier to pick up as TS (and other related languages like JavaScript) have either been widely adopted by teams or are relatively easy to pick up for even inexperienced developers.

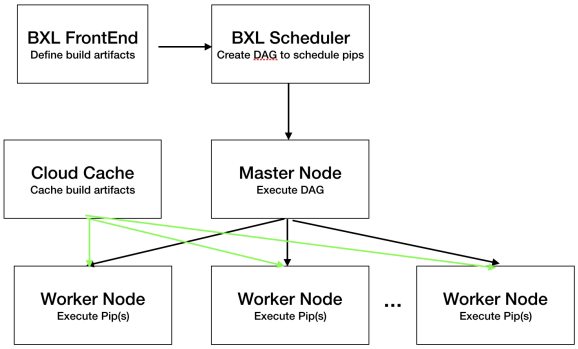

Figure 1: BXL system overview. Log files are generated at each node and then sent back to the master where they are combined together and then compressed for long term storage.

2.1.2 BXL Back-End.

The C# back-end's first job is to take the compiled DScript output and turn it into a DAG (directed acyclic graph). The generated build graph can be quickly compared with graphs from previous builds to determine similarity and be checked with the cache (more on it below) to determine how much actually needs to be built. This feature is also known as incremental building. Of course, if the cache is empty or nothing has been built in the past, the entire build graph is executed. Users can specify the number of machines that will be used, and the graph is distributed across the machines appropriately.

The second job of the backend is to actually coordinate, execute, and log the distributed build. The tasks (called pips) are sent to machines to be executed (if they have not been cached) and then as the pips are completed, various stats are logged. One machine serves as the master node and is in charge coordinating and delegating the work to worker nodes. Some log data is kept locally on worker node, and some is sent back to the master node via RPC requests. Eventually, after the build is done, the worker nodes send back all local data to the master node, and the master node writes the build graph into the log as well. These logs are compressed and stored for various times per the customers' requests.

2.1.3 BXL Cache.

The caching layer of BuildXL is arguably one of the most important layers as it in charge of deciding whether a pip (task) must be executed or if it can be fetched from the cache.

The cache can be run locally on a local machine, or can be set up as a "dev" cache that runs on remote machines. The default choice is the local cache which is used by most developers, but the larger builds on dedicated machines are configured for the dev cache. The dev cache additionally has multiple layers of caching that can connect to outputs from various datacenter tools (ie. CloudBuild) and also enables peer-to-peer caching at high network speeds. The cache layer is built on top of RocksDB, a persistent log-structured key-value store, where the keys are hashes (fingerprints) of various pips and inputs/outputs, and the values are the payloads themselves.

The caching layer uses a form of two-phase cache lookup pattern to first probe the cache, using the hashed fingerprint of the pip or artifact in question, for a small subset of possible outputs. This small subset can then be further narrowed down for all or most of the files and directories that are required. In the case that everything is found, then it is considered a "cache hit", but if it is a cache miss, then that pip is re-executed and its new contents are hashed and stored in the caching layer. A further, more in-depth explanation of the cache system can be found at this link: https://github.com/microsoft/BuildXL/blob/master/Public/Src/Cache/README.md.

2.1.4 BXL Log Analyzers.

The final component of BuildXL are the log file analyzers. The initial improvements from 90+ hours down to 10-12 hours was nearly immediately observable through the distribution of the build, but getting several percent speedup every quarter after (a common goal) requires much more effort and digging. The best place to dig are the massive logs generated by the build, and as such, some analyzers were created to crawl through the logs. These analyzers ranged from small queries such as "How much time did pip XYZ take to execute?" to more complicated queries like "What are the the top N dependency chains that take the most amount of time?" and more. The analyzers are also written in C# and developers must use the APIs provided by the BXL framework in order to crack open the logs for analysis. The logs and the analyzers will be the focus of the rest of this report.

2.2 Logs

As aforementioned, the logs created by a BuildXL build are critical in determining regressions in build times, anomalies, and in general any metric about the build. Teams were becoming more dependent on the logs as each quarter went by, since analyzing and mining the information here is one of the most assured methods of getting build speedups and insights. However, there were a myriad of problems with the current logging system that were becoming a nuisance for both BXL team members, and other developers across Microsoft. Notable these were:

(1) To save on space, logs were written in a custom binary format that packed all information as closely as possible without any padding in between [4]. The log elements contained a small header that told the analyzers how many bytes the element was, and what kind of element it is (so it can be read in appropriately). Any changes to element contents requires the custom serializer and deserializer to be updated, which can be overlooked. Furthermore, documentation may not be updated to show the latest logging information.
(2) The previous problem is accentuated by the fact that analyzers require the custom BXL API to crack open logs. However when the ser/deser of the elements are changed, analyzers using the old BXL API can no longer open newer logs. The API is not packaged in a Nuget package, but rather requires the user to build from source. This means that either developers need multiple versions of BXL on their systems to analyze logs across many months (or even weeks if many changes occur in a short time frame). A clean rebuild to an earlier version can take anywhere between ten minutes to one hour depending on the performance of the developer's workstation. This is a nuisance that has been more frequently encountered by developers. It is particularly a problem when regression testing/debugging is underway.
(3) The current format of the logs and analyzers were also not performant at all. For small local builds, most analyzers would take on the order of 30 seconds to a couple of minutes, but for large builds, analyzers can take anywhere from 10 to 15 minutes (and more) for complex analysis. The reason behind this performance is that the logs must be read in each time an analyzer is invoked, and the build graph and all dependencies are loaded back into memory. This operation itself can take up 50 - 75 % of the analysis time. Furthermore if two back-to-back analyzers are run, everything must once again be loaded back into memory, causing developers to become quickly frustrated if a single (or several) queries have yet to yield any results.
(4) The final issue with the current analyzers is that they are not platform agnostic. They are heavily tied to the BXL API which is exclusively in C#, and thus all anaalyzers must also be written in C#. This may not have been a problem before when much of Microsoft revolved around the .NET frameworks, but increasingly more developers are exclusively using languages

like Python, C++, or Go and as such they would prefer being able to crack open the logs and analyze them in their own language.

The four issues presented above, in rough order of importance, were the basis of my internship project and reason that Execution Log DataBase, Xldb, was created.

3 XLDB SYSTEM OVERVIEW

Xldb can best be described as a distributed logging database that allows for fast insertions during build time, and extremely fast lookup during analysis time. This section has been segmented into three parts. The first is what I was able to work on during my internship, and includes the bulk of the initial design and naive implementation of Xldb. The next two are logical partitions/continuations of the work after the internship finished.

3.1 Xldb V0/V1

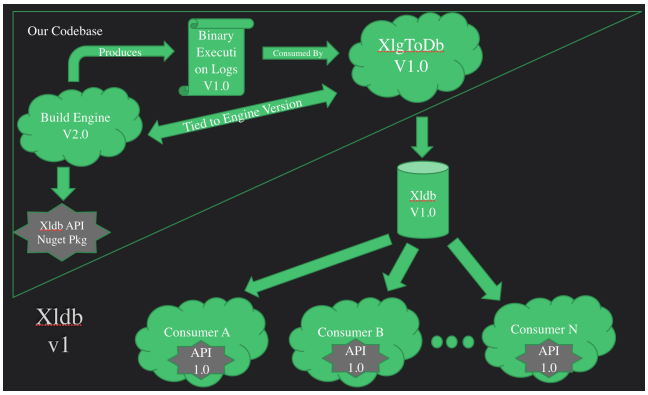

Figure 2: Xldb v1 system overview. In addition to the Nuget pkg API, a Python or C++ API could also be used with relative ease.

Xldb consists of two main parts: the underlying database, and the accessor API. Both of these were carefully picked, and designed to mitigate the performance issues of past logs/analyzers, and to address the compatibility issues when the log formats were updated. As mentioned earlier, this version of Xldb was a naive proof of concept and culminated with users having the opportunity to transform all old log formats in to the new Xldb format for future use. This involves running an "XlgToDb" program which is still tied to engine versions, but produces a log format that is compatible from here on out. Nonetheless, the design and overall idea are critical for future versions of the system. This initial version resulted in 5x - 200x speedup in analyzers while incurring a cost of 2x - 3x more storage. At the time, this was considered performant enough since the main goals were to untie the logs from the engine and enable forwards/backwards compatibility.

3.1.1 *Underlying DB.*

To speed up accesses to the logs, I needed a way to enable random or pseudo-random access. I initially tried creating a separate index file that could be crawled to look for offsets in the binary format that contained the data of interest, but soon realized that this would lead to the same breakage experienced in the old log format. Every time the underlying binary format was changed (when fields were added or removed), the indexing and offsets would have to be recalculated. This further results in the crawler for the index file to have to be tied to the BXL API and be "smart" about all of these changes. Thus I scrapped this approach and decided to go with a database instead.

I looked into several choices of databases including document based DBs like LiteDB and MongoDB, a traditional SQL DB (MariaDB) and NoSQL DBs like RocksDB. I ran some initial performance tests on these DBs by injecting dummy data to simulate various sized builds. The results indicated that though I could use BSON (Binary JSON) to concisely store data and also index into it for both LiteDB and MongoDB, the size of the DB exceeded the original compressed binary log format by more than 10x. Similarly, the MariaDB tests showed that size was growing beyond a reasonable bound as the size of the simulated builds increased, and the data was not able to be effectively split into multiple columns to utilize joins.

The simulated results for RocksDB proved to be quite interesting however [2]! The Cache layer of BXL already used RocksDB for storing the cached computations and treated the fingerprints as the keys and the objects as the values. I did not have any such notion for the logs so for the initial testing, I just using numerically ascending values as the keys and the different log objects for the values. The overall storage increase was only about 2x - 3x over the original binary format. This substantially lower storage gain comes from RocksDB's ability to perform compactions over the stored data to reduce the overall storage size. While the overall storage space can be reduced further, at the moment for v0/1, this was enough. There is likely no way to reduce it further the original specialized binary format, so that will remain the baseline for the foreseeable future.

The next step in the DB design was picking appropriate keys to fully utilize the key-value nature of RocksDB. The stored values needed to be forwards/backwards compatible and so they were designed to be ProtoBuf V3 messages with an official schema that could be carefully evolved to not break previous formats [3] [5]. This included relying on the ProtoBuf compiled parsers that could safely ignore missing

Default	Pip	Event	StaticGraph

Table 1: Original column families of Xldb (v0/v1). The default column is used to hold all "random" metadata that does not fit under one of the three other column families.

or additional fields that were not in the schema. RocksDB by default allows for precise key-value lookups, but the BXL Cache team had also created a "prefix-search" capability to allow lookups for all keys that matched a particular prefix. Thus the keys were also designed to be ProtoBuf V3 messages, with each value ordered from the most general field to more specific fields. The keys were also flattened so that ProtoBuf would not insert field binary messages (for nested messages, ProtoBuf inserts the lengths of the serialized nested message as well).

The final detail about the DB is the use of column families. Like tables in a traditional DB, the use of column families allows the data to partitioned in a logical manner, and also allows the RocksDB indexing and compactions to be done on a column family basis (better locality). The downside is that each column family can add up to 70 MB of overhead (in the worst case), but this is dwarfed by the many tens of gigabytes of data that is inserted into the DB by sufficiently large builds. The column families used split the data between Build Graph data, metadata, and other event data that was emitted during the build.

3.1.2 Accessor API.

The final piece to this puzzle is the accessor API for the DB. The main goal here was to make it simple enough to be emulated in multiple programming languages - anything that supports RocksDB and Protobuf V3 - yet also be as rich as the old accessor API. The API was initially developed in C#, but rigourously documented so it could be transferred into any supported language with ease. It extensively uses the notion of "prefix searches" to grab all possible related objects, before applying additional filters to match what the user may be interested in. The API was also deliberately left to provide only high level filtering, and requires the users to do additional filtering on objects on the receiving end. While objects themselves maintain compatibility through proper schema evolution, since the keys in the DB are also ProtoBuf V3 schemas, the API does face the possibility of falling out of sync, particularly if fields are changed or different column families are added. To mitigate this situation, two constraints have been created. First, fields can only be appended to a message in ascending order, and second, a separate text file would be added to the the folder containing the DB that specified the DB version. If the API version is greater than or equal to the DB version, it can successfully access the data, else it cannot and an older API will need to be generated. One may think that this latter situation leads to the same issues as before where an out of sync accessor API needs to be brought back in sync with a rebuild of the codebase. However, the current API is not tied to the engine version, only to the schemas of the various ProtoBuf messages, *and* the API is published as packages (whether it be Python, C# or otherwise). It is far easier to download a small package (or to keep several copies locally) than it is to rebuild the whole BXL codebase from scratch.

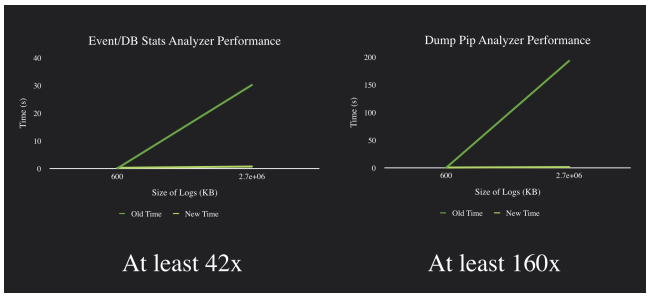

Figure 3: A comparison of two analyzers that have been ported over to the new format. Xldb performance grows exponentially as the size of the log files grow!

3.2 Xldb V2

While v0/v1 were a product of my internship at Microsoft, the overall system itself was open sourced and I decided to continue working on it for my capstone. My team and I had come up with a few other "future" work milestones that would be tackled long after my internship ended. I consider v2 of Xldb to be the addition of the following two features: a new organization of the underlying DB to compact the paths and strings, and a new Python API that can be used by developers. The addition of the string/path compaction results in the analyzer speedup (relative to the old analyzers before Xldb) to be reduced to 3x - 150x, but it also reduces the storage footprint to 1.5x - 2.5x which can result in many gigabytes saved per build (terabytes per day, relative to the original design)!

3.2.1 Path and String Compaction.

The first feature was a revamp of the database to utilize less storage. In the overall system, BuildXL, paths and string constants can take up the most amount of storage (both memory wise and disk storage in logs). This is primarily due to the massive number of string constants associated with

pips and paths associated various files and directories. Most of the time, these values occur more than once, so like how we can reuse objects or threads in an object pool/thread pool respectively, the decision to create a "pool" for the values was made. The results are the StringTable and PathTable constructs which at a high level act like a key-value map: each string constant is mapped to some integer constant, and that integer is used everywhere that is needed. The space savings were phenomenal, anywhere between 10% and 60%, depending again on the size and type of build.

Default	Pip	Event	StaticGraph	StringTable	InverseStringTable	PathTable	InversePathTable

Table 2: New column families of Xldb (v2 and beyond). The StringTable and PathTable column families map integers to strings, while the InverseStringTable and InversePathTable column families map strings to integers.

A similar strategy was employed in the restructuring of the database for v2 of Xldb. Once again, the paths and string constants were being repeated in many places when they could be amortized via a mapping. As the old log files are crawled and converted, the string and path table maps are constructed. Unfortunately due to the design of the original classes, we cannot just traverse them separately, and so this in-memory map must be utilized to hold the intermediary values. After all the events and the static graph are ingested, the two maps are written to the database. Four new column families were utilized: StringTable, InverseStringTable, PathTable, and InversePathTable (2). All four column families use ProtoBuf messages for both the keys and the values. The ordinary column families map from an integer to the string value, while the inverse column families map from string to an integer. By using a ProtoBuf message in the inverse column families, we can still employ the prefix-search API to search for the integer representations of all strings of a certain prefix (the empty string of course will match all keys in the prefix search).

This change also required the accessor API to be changed since several of the keys that used to be strings were now ints instead. Furthermore, every value that used to store a string now stores an int, so if a user wants to get the actual path or string representation, they need to make another call to the API to resolve the value. Unfortunately, even though RocksDB has sub millisecond latency for accessing values, this added complexity for every string does incur a slowdown for analyzers. With testing several different analyzers on a variety of builds, I saw only a speedup of 3x - 150x relative to the old (pre Xldb v0) log analyzers. However, to alleviate this issue, resolving the string values has been left as an optional task for the user. If they do not care about the representation for their particular analyzer, they need not resolve it, or if they only care about a subset of all constants, they can choose to just resolve those constants. This flexibility allows the resulting DB change to maintain the old speed in the case that no strings are resolved. However the more important advantage with this new format is the space savings. Switching to this DB format resulted in anywhere between 8% and 30% storage savings, resulting in the overall footprint to be only between 1.5x - 2.5x larger than the original log files!

3.2.2 Python API.

C# may still be one of the most popular languages to use at Microsoft (no licensing fees, in house experts, and more!), the engineer and developer community has begun to adopt other industry standards, including the widely popular language, Python. BuildXL is largely written in C# and since all of the old analyzers were tied to the engine, they were also written in C#. Xldb broke this tie in, yet the new accessor API for Xldb v0/v1 was also written in C# to show how old analyzers could be converted seamlessly.

However, now that the system has been in place for a couple of months, I believed it was time a new API binding to be introduced. I chose Python since it is one of the easier languages to pick up by students and hardened engineers alike, and it fully supports RocksDB and ProtoBuf, two of the main requirements of Xldb. The Python API also contains a version of the prefix-search capability that the C# API contains, but it is even more naive and thus can incur heavy penalties if the wrong prefix searches are initiated (ie. a full prefix search with the empty string ""). However in terms of raw performance, the Python RocksDB module is written in Cython and thus hooks right onto the C++ implementation of RocksDB. Thus while one may initially assume the performance suffers due to being written in Python relative to a compiled language like C#, the Python API is still blazing fast! One disadvantage though is that unlike the C# API which was published as a Nuget package (internal ring only for now), the Python module is not published anywhere since I no longer have access to the internal Microsoft module repository. Another minor inconvenience is that the Python proto files have to be slightly modified (flattened to one directory structure and remove the ‘/tools/‘ directory in front of Google packages), so I have written a helpful script to copy over the original files and modify them on the fly!

3.3 Xldb V3

I concluded my current work on the project with v3 of Xldb. Previous versions of Xldb still required manual initiation to convert old log files into the new log format and this

is quite a hassle for developers (and is also bad from an adoption perspective since there is a reduced ease of use for the new product). Since Xldb was still in a rather beta stage during my internship, we hesitate on providing an automatic conversion tool since there were of course legacy pipelines and work methodologies that were heavily reliant on the old log format. However now that it has been almost five months since the internship ended, it is time to auto-convert the logs.

While I could have taken a more naive approach to this problem and just created a wrapper script that does the conversion, this is ultimately another aspect of the codebase that must be maintained and is prone to being broken (shell scripts or batch files are often the most neglected and hardest to maintain pieces of code per my past experiences and experiences of former team members). Instead the implementation the automatic converter went through the existing BXL codebase and BXL scheduler. A flag (convertXldb) can be passed in that lets the scheduler know to use the new types of logs. Once the build is finished running and the old log format is created, the XlgToDBAnalyzer automatically kicks in and begins converting the logs in to the new Xldb format. This can take anywhere between a few seconds to up to 10 minutes, depending on the size of the logs and the type of machine being used. A server grade machine can easily take advantage of the parallelism in the analyzer and crunch through the conversion at ease! Once the logs have been converted, the old logs are deleted, except for the text based files which are useful for easy human readability. When the time of large builds can be on the scale of hours or tens of hours, a few additional minutes for converting the logs is a minor overhead for better accessibility of the log data.

4 DISCUSSION AND FUTURE WORK

The previous 3 versions (4 if you count v0) of Xldb included the prototyping stage and then the "flushing" stage where the system was modified per the customers' demands. The end result is phenomenal but it is not quite complete and there is still work to be done! Unfortunately I no longer have access to many of the resources that I did whilst an intern and development has come to a standstill. This includes powerful machines to speed up the build process, testing pipelines and builds in various canary rings for dogfooding, and ultimately quick access to my former team to get quick feedback and throw my ideas around. However, *should* I have had these resources, these are the following two versions of Xldb that I propose for future work to make it an even *more* powerful tool for developers using the Bxl system.

4.1 Xldb V4

V4 of Xldb aims to do what v3 partially attempted: automatic creation of the new Xldb logs. However instead of being a stage that happens after the build is finished, a more nuanced approach would be to ditch the old log format entirely and only write out Xldb formatted logs. There are a couple reasons that this is difficult to achieve apart from not having the right kind of resources as aforementioned, namely the backwards compatibility for teams that are still reliant on the old log format for legacy pipelines and due to the performance impact on the build itself.

The old log format creates a separate log on each machine in the build and then combines it all in the master node. One may think that this approach will also work in Xldb, but since Xldb creates many intermediate dictionaries to explicitly write information in a way that speeds up queries (ie. to speed up the cache miss analyzers or the input/output dependency chain analyzers), the task becomes more difficult. Some initial testing suggests that these dictionaries will either need to be synchronized via a cloud version of RocksDB (ie. like how the caache is syncrhonized) or it will require message passing between the nodes via GRPC calls.

4.2 Xldb V5: XaaS

With the full indepedence of logs from the engine and an easy to use API that is versatile across many languages and platforms, I think the logical evolution of Xldb is to present it as a service: Xldb as a Service (XaaS). Instead of having individual developers download logs to their machines and analyzing the logs, the team could instead have ownership over all logs for a certain date (a month or two perhaps) and allow others to query the logs remotely. This drastically reduces the amount of network bandwidth used for downloading all of the logs and also opens up the possibility for a few other interesting features:

(1) Queries can be sent in like "jobs" which can be run simultaneously over multiple logs. The results can be aggregated and delivered back to the developer.
(2) A dashboard can be created for CB/CQ (continious build, continuous querying) purposes.
(3) Triggers can be created where if a build performs a certain "trigger" action, it can automatically run queries on the logs (in a fast manner relative to today) and send the results to the appropriate developers.
(4) Regressions can be better tracked by both the BXL team as well as other build engineers.
(5) Provide a log ingestion method for users to ingest logs in their own custom programs or frameworks automatically.

5 CONCLUSION

Codebases continue to get larger and more complex as customers demand more, and features are supported for longer amounts of time. Microsoft is no stranger to this issue and to address the long build times, they created a distributed build tool, BuildXL. Developers became increasingly interested in crawling and mining the logs produced but as the logs became more complex, they began to break the custom binary format more often, and were slow to traverse. Xldb was created to counter these issues by allowing for full forwards/backwards compatibility and faster querying. It achieves these properties through a RocksDB instance that holds ProtoBuf v3 keys and values, and an API that can be implemented in any language that supports RocksDB and ProtoBuf. The resulting system is 3x - 150x faster for most queries (at least 80% of the popular queries), and the performance gains increase as the size of the build increases. The storage takes a hit as Xldb requires 1.5x - 2.5x more storage than the previous format, but it also allows the logs to be platform independent so more developers can take advantage of them. Currently production software can use v1 of Xldb, with future versions either merged into the canary rings or (potentially) waiting to be merged in the near future. The official repository is located at https://github.com/microsoft/BuildXL and my capstone work can be found in the following fork: https://github.com/sahilmgandhi/BuildXL/tree/capstone-work.

ACKNOWLEDGMENTS

I want to acknowledge Oleksii Kononenko for mentoring me during the internship, and Mike Pysson for being a great manager. I also want to thank Lance Collins and Danny Van Velzen for helping out immensely in designing and working with the front end (dscript) parts of BuildXL. They were great mentors and team members throughout my internship and even as I worked on the project afterwards.

REFERENCES

[1] Apache. [n. d.]. Apache Avro. https://avro.apache.org/
[2] Facebook. [n. d.]. A persistent key-value store. https://rocksdb.org/
[3] Google. [n. d.]. Google Protocol Buffers. https://developers.google.com/protocol-buffers
[4] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392
[5] Kåre Kjelstrøm, Kåre Kjelstrøm, Kare, Kare, Uber, and Uber's Core Infrastructure. 2018. How Uber Engineering Evaluated JSON Encoding and Compression Algorithms to Put the Squeeze on Trip Data. https://eng.uber.com/trip-data-squeeze/
[6] Microsoft. 2020. microsoft/msbuild. https://github.com/microsoft/msbuild