
# INSTalytics: Cluster Filesystem Co-design for Big-data Analytics

**Muthian Sivathanu, Midhul Vuppalapati, Bhargav Gulavani, Kaushik Rajan, and Jyoti Leeka,** ***Microsoft Research India;*** **Jayashree Mohan,** ***Univ. of Texas Austin;*** **Piyus Kedia,** ***IIIT Delhi***

https://www.usenix.org/conference/fast19/presentation/sivathanu

**This paper is included in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19).**

**February 25–28, 2019 • Boston, MA, USA**

978-1-931971-48-5

**Open access to the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19) is sponsored by**

NetApp®

# INSTalytics: Cluster Filesystem Co-design for Big-data Analytics

Muthian Sivathanu*, Midhul Vuppalapati*, Bhargav S. Gulavani*, Kaushik Rajan*,
Jyoti Leeka*, Jayashree Mohan†, Piyus Kedia◇*

*Microsoft Research India, †Univ. of Texas Austin, ◇IIIT Delhi

## Abstract

*We present the design, implementation, and evaluation of INSTalytics a co-designed stack of a cluster file system and the compute layer, for efficient big data analytics in large-scale data centers. INSTalytics amplifies the well-known benefits of data partitioning in analytics systems; instead of traditional partitioning on one dimension, INSTalytics enables data to be simultaneously partitioned on four different dimensions at the same storage cost, enabling a larger fraction of queries to benefit from partition filtering and joins without network shuffle.*

*To achieve this, INSTalytics uses compute-awareness to customize the 3-way replication that the cluster file system employs for availability. A new heterogeneous replication layout enables INSTalytics to preserve the same recovery cost and availability as traditional replication. INSTalytics also uses compute-awareness to expose a new* sliced-read *API that improves performance of joins by enabling multiple compute nodes to read slices of a data block efficiently via co-ordinated request scheduling and selective caching at the storage nodes.*

*We have built a prototype implementation of INSTalytics in a production analytics stack, and show that recovery performance and availability is similar to physical replication, while providing significant improvements in query performance, suggesting a new approach to designing cloud-scale big-data analytics systems.*

## 1 Introduction

*All the powers in the universe are already ours. It is we who put our hands before our eyes and cry that it is dark.*

*- Swami Vivekananda*

Large-scale cluster file systems [10, 22, 17] are designed to deal with server and disk failures as a common case. To ensure high availability despite failures in the data center, they employ *redundancy* to recover data of a failed node from data in other available nodes [11]. One common redundancy mechanism that cluster file systems use for compute-intensive workloads is to keep multiple (typically three) copies of the data on different servers. While redundancy improves availability, it comes with significant storage and write amplification overheads, typically viewed as the *cost* to be paid for availability.

An increasingly important workload in such large-scale cluster file systems is big data analytics processing [6, 31, 2]. Unlike a transaction processing workload, analytics queries are typically scan-intensive, as they are interested in millions or even billions of records. A popular technique employed by analytics systems for efficient query execution, is partitioning of data [31] where the input files are *sorted* or *partitioned* on a particular column, such that records with a specific range of column values are physically clustered within the file. With partitioned layout, a query that is only interested in a particular range of column values (say 1%) can use metadata to only scan the relevant partitions of the file, instead of scanning the entire file (potentially tens or hundreds of terabytes). Similarly, with partitioned layout, join queries can avoid the cost of expensive network shuffle [29].

However, as partitioning is tied to the physical layout of bytes within a file, data is partitioned only on a single dimension; as a result, it only benefits queries that perform a filter or join on the column of partitioning, while other queries are still forced to incur the full cost of a scan or network shuffle.

In this paper, we present *INSTalytics* (INtelligent STore powered Analytics), a system that drives significant efficiency improvements in performing large-scale big data analytics, by amplifying the well-known benefits of partitioning. In particular, *INSTalytics* allows data to be partitioned simultaneously along *four* different dimensions, instead of a single dimension today, thus allowing a large fraction of queries to achieve the benefit of efficient partition filtering and efficient joins without network shuffle. The key approach that enables such improvements in *INSTalytics* is making the distributed file system *compute-aware*; by customizing the 3-way replication that the file system already does for availability, *INSTalytics* achieves such heterogeneous partitioning without incurring additional storage or write amplification cost.

The obvious challenge with such *logical replication* is ensuring the same availability and recovery performance as physical replication; a naive layout would require scanning the entire file in the other partitioning order, to recover a single failed block. *INSTalytics* uses a novel layout technique based on

*Jayashree and Piyus worked on this while at Microsoft Research

*super-extents* and *intra-extent circular buckets* to achieve recovery that is as efficient as physical replication. It also ensures the same availability and fault isolation guarantees as physical replication under realistic failure scenarios. The layout techniques in *INSTalytics* also enable an additional fourth partitioning dimension, in addition to the three logical copies.

The file system in *INSTalytics* also enables efficient execution of join queries, by supporting a new *sliced-read* API that uses compute-awareness to co-ordinate scheduling of requests across multiple compute nodes accessing the same storage extent, and selectively caches only the slices of the extent that are expected to be accessed, instead of caching the entire extent.

We have implemented *INSTalytics* in a production distributed file system stack, and evaluate it on a cluster of 500 machines with suitable modifications to the compute layers. We demonstrate that the cost of maintaining multiple partitioning dimensions at the storage nodes is negligible in terms of recovery performance and availability, while significantly benefiting query performance. We show through micro-benchmarks and real-world queries from a production workload that *INSTalytics* enables significant improvements upto an order of magnitude, in the efficiency of analytics processing.

The key contributions of the paper are as follows.

- We propose and evaluate novel layout techniques for enabling four simultaneous partitioning/sorting dimensions of the same file without additional cost, while preserving the availability and recovery properties of the present 3-way storage replication;
- We characterize a real-world analytics workload in a production cluster to evaluate the benefit of having multiple partitioning strategies;
- We demonstrate with a prototype implementation that storage co-design with compute can be implemented practically in a real distributed file system with minimal changes to the stack, illustrating the pragmatism of the approach for a data center;
- We show that compute-awareness with heterogenous layout and co-ordinated scheduling at the file system significantly improves the performance of filter and join queries.

The rest of the paper is structured as follows. In § 2, we provide a background of big data analytics processing. In § 3, we characterizate a production analytics workload. We present logical replication in § 4, discuss its availability implications in § 5, and describe optimizations for joins in § 6. We present the implementation of *INSTalytics* in § 7, and evaluate it in § 8. We present related work in § 9, and conclude in § 10.

# 2 Background

In this section, we describe the general architecture of analytics frameworks, and the costs of big data query processing.

**Cluster architecture:** Big data analytics infrastructure typically comprises of a compute layer such as MapReduce [6] or Spark [28], and a distributed file system such as GFS [10] or HDFS [22]. Both these components run on several thousands of machines, and are designed to tolerate machine failures given the large scale. The distributed file system is typically a decentralized architecture [14, 26, 10], where a centralized "master" manages metadata while thousands of storage nodes manage the actual blocks of data, also referred to as chunks or extents (we use the term "extent' in the rest of the paper). An extent is typically between 64MB and 256MB in size. For availability under machine failures, each storage extent is replicated multiple times (typically thrice). The compute layer can run either on the same machines as the storage nodes (*i.e.*, co-located), or a different set of machines (*i.e.*, disaggregated). In the co-located model, the compute layer has the option of scheduling computation for locality between the data and compute, say at a rack-level, for better aggregate data bandwidth.

**Cost of analytics queries:** Analytics queries often process millions or billions of records, as they perform aggregation or filtering on hundreds of terabytes. As a result, they are scan-intensive on the disks - using index lookups would result in random disk reads on millions of records. Hence disk I/O is a key cost of query processing given the large data sizes.

An important ingredient of most big data workloads is *joins* of multiple files on a common field. To perform a join, all records that have the same join key value from both files need to be brought to one machine, thus requiring a *shuffle* of data across thousands of servers; each server sends a *partition* of the key space to a designated worker responsible for that partition. Such all-to-all *network shuffle* typically involves an additional disk write of the shuffled data to an intermediate file and subsequent disk read. Further, it places load on the data center switch hierarchy across multiple racks.

**Optimizations:** There are two common optimizations that reduce the cost of analytics processing: partitioning, and co-location. With partitioning, the data file stored on disk is sorted or partitioned by a particular dimension or column. If a query filters on that same column, it can avoid the cost of a full scan by performing *partition elimination*. With co-location, joins can execute without incurring network shuffle. If two files A and B are likely to be joined, they can both be partitioned on the join column/dimension in a consistent manner, so that their partition boundaries align. In addition, if the respective partitions are also placed in a rack-affinitized manner, the join can avoid cross-rack network shuffle, as it would be a per-partition join. Further, the partitioned join also avoids the intermediate I/O, because the respective partitions can perform the join of small buckets in memory.

# 3 Workload Analysis

We analyzed one week's worth of queries on a production cluster at *Microsoft* consisting of tens of thousands of servers. We

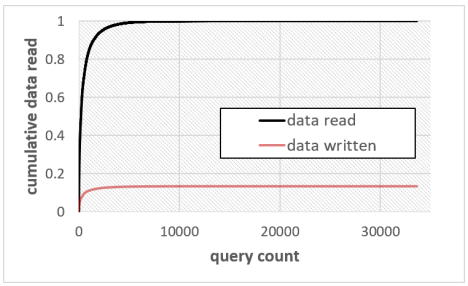


Figure 1: Data filtering in production queries

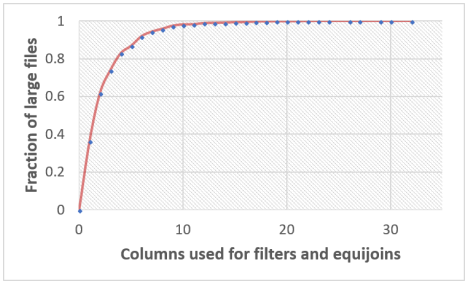


Figure 2: Number of partition dimensions needed.

share below our key findings.

### 3.1 Query Characteristics

**Data filtering:** The amount of data read in the first stage of the query execution vis-a-vis the amount of data written back at the end of the first stage indicates the degree of filtering that happens on the input. In Figure 1, we show a CDF of the data read and data written by unique query scripts (aggregating across repetitions) along the X-axis, sorted by the most expensive jobs. As the graph shows, on average, there is a 7x reduction in data sizes past the first stage. This reduction is due to both column-level and row-level filtering; while column-stores [15] help with the former, we primarily focus on row-level filtering which is complementary.

**Importance of join queries:** Besides filtering, joins of multiple files are an important part of big data analytics. In our production workload, we find that 37% of all queries contain a join, and 85% of those perform a join at the start of the query.

### 3.2 Number of dimensions needed

Today's systems limit the benefit of partitioning to just one dimension. To understand what fraction of queries would benefit from a higher number of dimensions, we analyzed each input file referenced in any job during a week, and we extracted all columns/dimensions that were ever used in a filter or join clause in any query that accessed the file. We plot a CDF of the fraction of files that were only accessed on K columns (K varying along X axis). As Figure 2 shows, with one partitioning dimension, we cover only about 33% of files, which illustrates the limited utility of today's partitioning. However, with 4 partitioning dimensions, the coverage grows significantly to about 83%. Thus, for most files, having them partitioned in 4 dimensions would enable efficient execution for all queries on that file, as they would benefit from partitioning or co-location.

**Supporting multiple dimensions:** Today, the user can partition data across multiple dimensions by storing multiple copies. However this comes with a storage cost: to support 4 dimensions, the user incurs a 4x space overhead, and worse, a 4x cost in write bandwidth to keep the copies up to date. Interestingly, in our discussions with teams that perform big-data analytics within *Microsoft*, we found examples where large product groups actually maintain multiple (usually 2) copies (partitioned on different columns) just to reduce query latency, despite the excessive cost of storing multiple copies. Many teams stated that more partition dimensions would enable new kinds of analytics, but the cost of supporting more dimensions today is too high.

## 4 Logical Replication

The key functionality in *INSTalytics* is to enable a much larger fraction of analytics queries to benefit from partitioning and co-location, but without paying additional storage or write cost. It achieves this by co-designing the storage layer with the analytics engine, and changing the *physical* replication (usually three copies that are byte-wise identical) that distributed file systems employ for availability, into *logical* replication, where each copy is partitioned along a different column. Logical replication provides the benefit of three simultaneous partitioning columns at the same storage and write cost, thus improving query coverage significantly, as shown in Section 3. As we describe later, our layout actually enables *four* different partitioning columns at the same cost.

The principle of logical replication is straight-forward, but the challenge lies in the details of the layout. There are two conflicting requirements: first, the layout should help query performance by enabling partition filtering and collocated joins in multiple dimensions; second, recovery performance and availability should not be affected.

In the rest of the section we describe several variants of logical replication, building towards a more complete solution that ensures good query performance at a recovery cost and availability similar to physical replication. For each variant, we describe its recovery cost and potential query benefits. We use the term *dimension* to refer to a column used for partitioning; each logical replica would pertain to a different dimension. We use the term *intra-extent bucketing* to refer to partitioning of rows within an extent; and we use the term *extent-aligned partitioning* to refer to partitioning of rows across multiple extents where partition boundaries are aligned with extent boundaries. When the context is clear, we simply use the terms *bucketing* and *partitioning* respectively for the above.

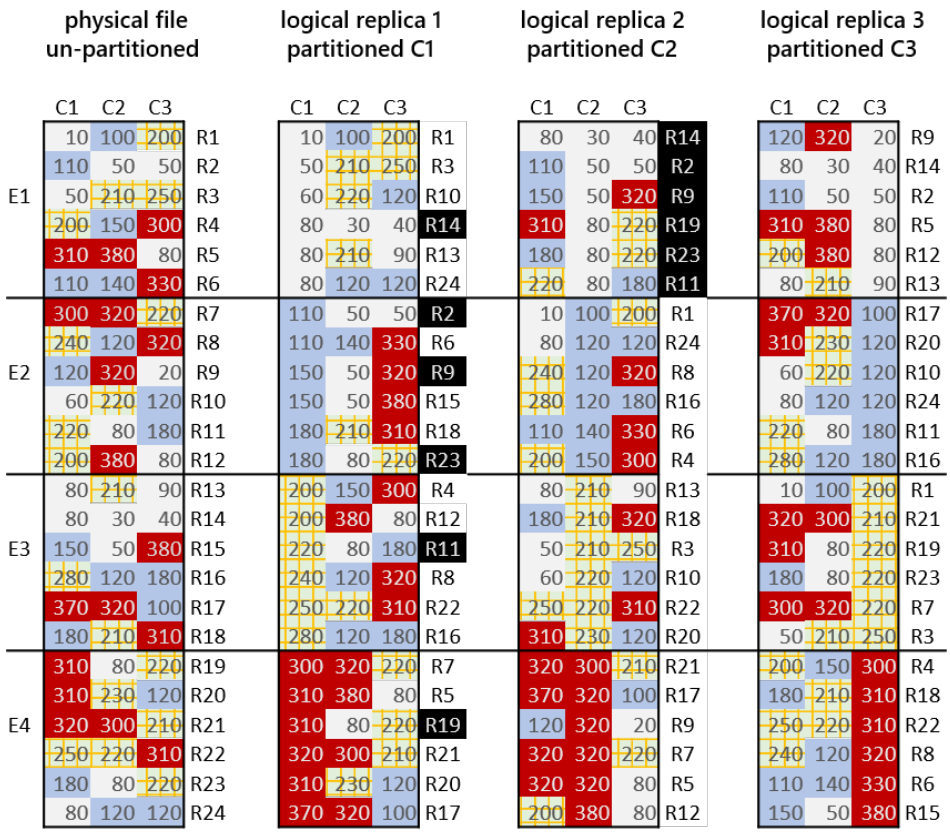


Figure 3: **Naive logical replication.** The figure shows the physical layout of the input file and the layout with naive logical replication. The file has 4 extents each consisting of 6 rows, 3 columns per row. Each logical replica is range partitioned on a different column. The first replica is partitioned on the first column (*e.g.*, E1 has values from 0-99, E2 from 100-199 and so on). Recovering E1 from replica 2 requires reading all extents from replica 1.

## 4.1 Naive layouts for logical replication

There are two simple layouts for logical replication, neither meeting the above constraints. The first approach is to perform logical replication at a file-granularity. In this layout the three copies of the file are each partitioned by a different dimension, and stored seperately in the file system with replication turned off. This layout is ideal for query performance as it is identical to keeping three different copies of the file. Unfortunately this layout is a non-starter in terms of recovery; as Figure 3 shows, there is *inter-dimensional diffusion* of information; the records in a particular storage extent in one dimension will be diffused across nearly all extents of the file in the other dimension. Thus, recovering a 200MB extent would require reading an entire say 10TB file in another dimension, whereas with physical replication, only 200MB is read from another replica.

The second sub-optimal approach is to perform logical replication at an intra-extent level. Here, one would simply use *intra-extent bucketing* to partition the records *within* a storage extent along different dimensions in each replica. This simplifies recovery as there is a one-to-one correspondence with the other replicas of the extent. This approach helps partly with filter queries as the file system can use metadata to read only the relevant buckets within an extent, but is not as efficient as the previous layout as clients would touch all extents instead of a subset of extents. The bigger problem though is that joins or aggregation queries cannot benefit at all, because co-location/local shuffle is impossible. We discuss more about the shortcomings of this approach in handling joins in Section 6.

## 4.2 Super-Extents

*INSTalytics* bridges the conflicting requirements of query performance and recovery cost, by introducing the notion of a *super-extent*. A super-extent is a fixed number (typically 100) of contiguous extents in the file in the original order the file was written (see Figure 4). Partitioning of records happens *only* within the confines of a super-extent and happens in an extent aligned manner. As shown in the figure this ensures that the inter-dimensional diffusion is now limited to only within a super-extent; all records in an extent in one dimension are hence guaranteed to be present somewhere within the corre-

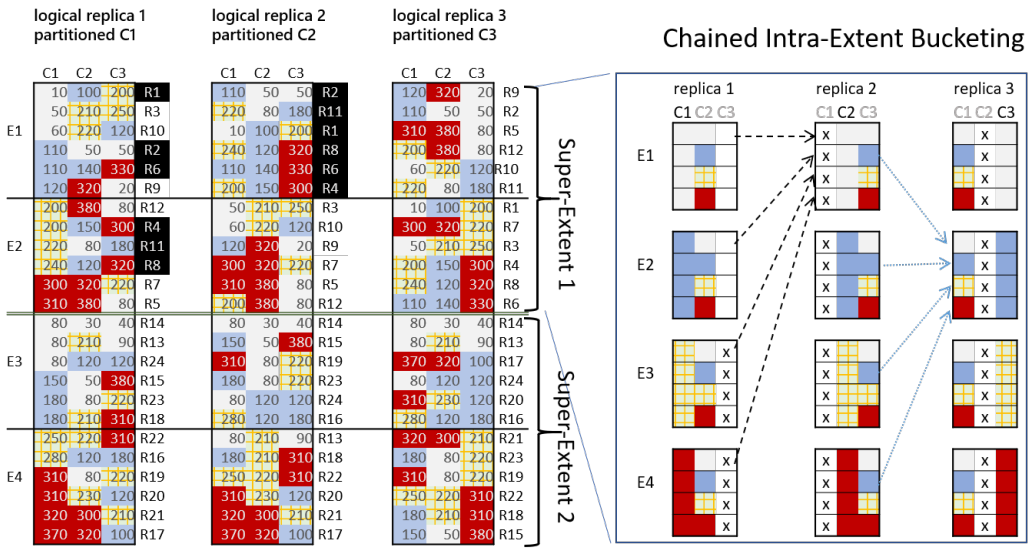


Figure 4: **Super-extents and intra extent bucketing.** The file to be replicated is divided into super extents and logical replication is performed within super-extents. Any extent (like E1 of replica 2 as highlighted) can be recovered by only reading extents of the same super extent in another replica (E1 and E2 from replica 1). Figure on the right shows a more detailed view of a super-extent (with 4 extents per super-extent instead of 2 for clarity). Data within an extent is partitioned on a different column and this helps reduce recovery cost further as recovering an extent only involves reading one bucket from each extent of the other replica. Recovering E4 of replica 2 requires only reading slices from replica 1 whose C2 boxes are red.

sponding super-extent (*i.e.*, 100 extents) of the other partitioning dimension. All 3 copies of the super-extent have exactly the same information, just arranged differently. The number of extents within a super-extent is configurable; in that sense, the super-extent layout can be treated as striking a tunable balance between the two extremes *i.e.*, global partitioning and intra-extent partitioning.

Super-extents reduce recovery cost because in order to recover a 200MB extent, the store has to read "only" 100 extents of that super-extent from another dimension. This is better than reading the entire say 10TB file with the naive apporoach, but the cost of recovery is still significantly higher (100x) than physical replication. We improve on this below. From a query perspective, the super-extent based layout limits the granularity of partitioning to the number of extents per super extent. This limits the potential savings for filtering and co-location. For example, consider a very selective filter query that matches only 1/1000th of the records. With the super extent layout, the query would still have to read one extent in each super-extent. Hence the maximum speedup because of partition elimination (with 100 extents per super-extent) is 100x, whereas the file-level global partitioning could support 1000 partitions and provide a larger benefit. However, the 100x speed up is significant enough that the tradeoff is a net win. Conceptually, the super extent based layout does a partial partitioning of the file, as every key in the partition dimension could be present in multiple extents, one per super extent. The globally partitioned layout would have keys more densely packed within fewer extents.

**Fourth partition dimension.** Finally, super-extents benefit query execution in a way that file-level global partitioning does not. As we do not alter the native ordering of file extents across super-extent boundaries, we get a fourth dimension. If the user already partitioned the file on a particular column, say timestamp, we preserve the coarse partitions by timestamp across super-extents, so a query filtering on timestamp can eliminate entire super-extents. Thus, we get 4 partition dimensions for no additional storage cost compared to today[1].

## 4.3 Intra-extent Chained Buckets

While super-extents reduce recovery cost to 100 extents instead of the whole file, the 100x cost is still a non-starter. Recovery needs to be low-latency to avoid losing multiple copies, and needs to be low on resource usage so that the cluster can manage recovery load during massive failures such as rack failures. Intra-extent chained buckets is the mechanism we use to make logical recovery as efficient as physical recovery.

The key idea behind intra-extent chained buckets, is to use bucketing *within* an extent for recovery instead of query performance. The records within an extent are bucketed on a dimension that is different from the partition dimension used within the super-extent. Let us assume that $C_1$, $C_2$, and $C_3$ are the three columns/dimensions chosen for logical replica-

[1]This fourth dimension is not equally powerful as the first three because while it provides partition elimintation for filters, it does not provide colocation for joins

tion. Given 100 extents per super-extent, the key-space of $C_1$ would be partitioned into 100 ranges, so the $i^{th}$ extent in every super-extent of dimension $C_1$ would contain records whose value for column $C_1$ fall in the $i^{th}$ range.

Figure 4 (right) illustrates how intra-extent chained buckets work. Let us focus on the first extent E1 of a super-extent in dimension $C_1$. The records within that extent are further bucketed by dimension $C_2$. So the 200MB extent is now comprised of 100 buckets each roughly of size 2MB. The first intra-extent bucket of E1 contains only records whose value of column $C_2$ falls in the first partition of dimension $C_2$, and so on. Similarly the extents of dimension $C_2$ have an intra-extent bucketing by column $C_3$, and the extents of dimension $C_3$ are intra-extent bucketed by column $C_1$.

With the above layout, recovery of an extent does not require reading the entire super-extent from another dimension. In Figure 4, to recover the last extent of replica 2, the store needs to read only the last bucket from extents E1 to E4 of replica 1, instead of reading the full content of those 4 extents.

Thus, in a superextent of 100 extents, instead of reading 100 x 200MB to recover a 200MB extent, we now only read 2MB each from 100 other extents, *i.e.*, read 200MB to recover a 200MB extent, essentially the same cost as physical replication. By reading 100 chunks of size 2MB each, we potentially increase the number of disk seeks in the cluster. However, given the 2MB size, the seek cost gets amortized with transfer time, so the cost is similar especially since physical recovery also does the read in chunks. As we show in Section 8, the aggregate disk load is very similar to physical replication. From a networking perspective, the bandwidth usage is similar except we have a parallel and more distributed load on the network.

Thus, with super-extents and intra-extent chained buckets, we achieve our twin goals of getting the benefit of partitioning and co-location for more queries, while simultaneously keeping recovery cost the same as physical replication.

## 4.4 Making storage record-aware

In order to perform logical replication, the file system needs to rearrange records across extents. However, the interface to the file system is only in terms of opaque blocks. Clients perform a *read block* or *write block* on the store, and the internal layout of the block is only known to the client. For example, the file could be an unstructured log file or a structured file with internal metadata. One could bridge this semantic gap by changing the storage API to be record-level, but it is impractical as it invovles changes to the entire software stack , and curtails the freedom of higher layers to use diverse formats.

To bridge this tension, we introduce the notion of *format adapters* in *INSTalytics*. The adapter is simply an encoder and decoder that translates back and forth between an opaque extent, and records within that extent. Each format would have its own adapter registered with the store, and only registered formats are supported for logical replication. This is pragmatic in cloud-scale data centers where the same entity controls both the compute stack and the storage stack and hence there is coordination when formats evolve.

A key challenge with the adapter framework is dealing with formats that disperse metadata. For example, in one of our widely-used internal formats, there are multiple levels of pointers across the entire file.there is a *footer* at the end of the file that points to multiple chunks of data, which in turn point to pages. For a storage node that needs to decode a single extent for logical replication, interpreting that extent requires information from several other extents (likely on other storage nodes), making the adapter inefficient. We therefore require that each extent is self-describing in terms of metadata. For the above format, we made small changes to the writer to terminate chunks at extent boundaries and duplicate the footer information within a chunk. Given the large size of the extent, the additional metadata cost is negligible (less than 0.1%).

## 4.5 Creating logical replicas

Logical replication requires application hints on the dimensions to use for logical replication, the file format, *etc.*. Also, not all files benefit from logical replication, as it is a function of the query workload, and the read/write ratio. Hence, files start off being physically replicated, and an explicit API from the compute layer converts the file to being logically replicated. Logical replication happens at the granularity of super-extents; the file system picks 100 extents, shuffles the data within those 100 (3-way replicated) extents and writes out 300 new extents, 100 in each dimension. The work done during logical replication is a disk read and a disk write. Failure-handling during logical replication is straight-forward: reads use the 3-way replicated copies for failover, and writes failover to a different available storage node that meets the fault-isolation constraints. The logical replication is not *in-place*. Until logical replication for a super-extent completes, the physically replicated copies are available, which simplifies failure retry. As the application that generates data knows whether to logically replicate, it could place those files on SSD, so that the extra write and read are much cheaper; because it's a transient state until logical replication, the SSD space required is quite small.

## 4.6 Handling Data Skew

To benefit from partitioning, the different partitions along a given dimension must be roughly balanced. However, in practice, because of data skew along some dimensions [4], some partitions may have more data than others. To handle this, *INSTalytics* allows for variable sized extents within a super-extent, so that data skew is only a performance issue, not a correctness issue. Given the broader implications of data skew for query performance, users already pre-process the data to

ensure that the partitioning dimensions are roughly balanced (by using techniques such as prefixing popular keys with random salt values, calibrating range boundaries based on a distribution analysis on the data, *etc.*). As the user specifies the dimensions for logical replication, as well as the range boundaries, *INSTalytics* benefits from such techniques as well. In future, we would like to build custom support for skew handling within *INSTalytics* as a more generic fallback, by performing the distribution analysis as part of creating logical replicas, to re-calibrate partition boundaries when the user-data is skewed.

# 5 Availability with logical replication

While the super-extent based layout ensures the same recovery cost as physical replication, it incurs a hit in availability. With physical replication, for an extent to become unavailable, all three machines holding the three copies of the extent must be unavailable. If $p$ is the independent probability that a specific machine in the cluster goes down within say a 5-minute window, the probability of unavailability in physical replication for a given extent is $p^3$. But with logical replication, an extent becomes unavailable if the machine with that extent is down, and additionally *any one* of the 100 machines in each of the other two dimensions containing replicas of the super-extent, are down, making the probability of unavailability $p \times 100p \times 100p = 10^4.p^3$. In this reasoning, we only consider independent machine failures, as correlated failures are handled below with fault-isolated placement.

## 5.1 Parity extents

To handle this gap in availability, we introduce an additional level of redudancy in the layout. Within a super-extent replica which comprises of 100 extents, we add *parity extents* with simple XOR parity for every group of 10 extents, *i.e.*, a total of 10 parity extents per super-extent replica. Now, each parity group can tolerate one failure, which means for unavailability, there has to be a double failure in the extent's parity group, and in addition, at least one parity group in each of the other two dimensions must have a double failure. The probability thus becomes $p.10p \times 10.\binom{11}{2}.p^2 \times 10.\binom{11}{2}.p^2 = 3.02 \times 10^6.p^6$. Solving this for $p$, as long as $(p < 0.7\%)$, the availability would be better than physical replication.[2]. . In rare cases of clusters where the probability is higher, we could use double-parity [5] in a larger group of 20 extents, so that each group of 20 extents can tolerate two failures. The probability of unavailability now becomes $p.\binom{21}{2}.p^2 \times 5.\binom{22}{3}.p^3 \times 5.\binom{22}{3}.p^3 = 12.45 \times 10^9.p^9$, so the cut-off point becomes $p < 2.1\%$. Note that another knob to control availability is the size of a super-extent: with super-extents comprising 10 extents instead of 100, the single parity itself can handle a significant failure probability of $p < 3.2\%$.

The machine failure probability $p$ above refers to the failure probability within a small time window, *i.e.*, the time it takes to recover a failed extent. This is much lower than the average % of machines that are offline in a cluster at any given time, because the latter includes long dead machines, whose data would have been recovered on other machines anyway. As we show in Section 8, this failure probability of random independent machines (excluding correlated failures) in large clusters is less than 0.2%, so single parity is often sufficient, and hence this is what we have currently implemented.

Recovering a parity extent requires 10x disk and network I/O compared to a regular extent, because it has to perform an XOR of 10 corresponding blocks. As 10% of logically replicated extents are parity extents, this would double the cost of recovery (10% x 10x). We therefore store two physically replicated copies of each parity block, so that during recovery, most of the failed parity blocks can be recovered with a raw copy, and we incur the 10x disk cost only for a tiny fraction. This is a knob for the cluster administrator - whether to incur the additional 10% space cost, or the 2x performance cost during recovery; in our experience, recovery cost is more critical and hence the default is 2-way replication of parity blocks.

## 5.2 Fault-isolated placement

Replication is aimed at ensuring availability during machine failures. As failures can be correlated, *e.g.*, a rack power switch can take down all machines in that rack, file systems perform fault-isolated placement. For example, the placement would ensure that the 3 replicas of an extent are placed in 3 different failure domains, to aovid simultaneously losing multiple copies of the block. With logical replication, each extent does not have a corresponding replica, thus requiring a different strategy for fault-isolation. The way *INSTalytics* performs this fault isolation is to reason at the super-extent level, because all replicas of a given super-extent contain exactly the same information. We thus place each replica of the super-extent in a disjoint set of failure domains relative to any other replica of the same super-extent, thus ensuring the same fault-isolation properties as physical replication.

# 6 Efficient Processing of Join Queries

The multi-dimensional partitioning in *INSTalytics* is designed to improve performance of join queries in addition to filter queries. In this section, we first describe the *localized shuffle* that is enabled when files are joined on one of the dimensions of logical replication. We then introduce a new compute-aware API that the file system provides, to further optimize joins by

[2]This formula is an approximation (for ease of understanding) that works for small values of $p$. The accurate (and more complex) formula is $p.(1-(1-p)^{10})(1-(1-p)^{100}(1+10p)^{10})^2$ which would be greater than $p^3$ (physical replication) as long as $p < 0.725\%$

completely eliminating network shuffle.

### 6.1 Localized Shuffle

Joins on a logically replicated dimension can perform *localized shuffle*: partition *i* of the first file only needs to be joined with partition *i* of the second file, instead of a global shuffle across all partitions. Localized shuffle has two benefits. First, it significantly lowers (by 100x) the fan-in of the "reduce" phase, eliminating additional intermediate "aggregation" stages that big-data processing systems introduce just to reduce the fan-in factor for avoiding small disk I/Os and for fault-tolerance [29]. Elimination of intermediate aggregation reduces the number of passes of writes and reads of the data from disk. Second, it enables network-affinitized shuffle. If all extents for a partition are placed within a single rack of machines, local shuffle avoids the shared aggregate switches in the data center, and can thus be significantly more efficient.

The placement by the file system ensures that extents pertaining to the same partition across all super-extents in a given dimension are placed within a single rack of machines. The file system supports an interface to specify during logical replication the logical replica of another file to co-affinitize with.

### 6.2 Sliced Reads

Localized shuffle avoids additional aggregation phases, but still requires one write and read of intermediate data. If the individual partitions were small enough, the join of each individual partition can happen in parallel in memory, avoiding this disk cost. However, as super-extents limit the number of partitions to 100, joins of large files (*e.g.*, 10 TB) will be limited by parallelism and memory needs at compute nodes.

To address this limitation, *INSTalytics* introduces a new file system API called a *sliced-read*, which allows a client to read a small semantic-slice of an extent that belongs to a sub-range of a partition, further sub-dividing the partition into 100 (*slice-factor*) buckets. For instance, if say the first (out of 100) partition represents the range 0-10k, a *sliced-read* can ask for only records in the range 9k-9.1k, thus providing the equivalent benefit of having 10,000 partitions on the original file while the super-extent remains at 100 partitions. Each compute node would now read one bucket from each super-extent. The ability to perform per-partition joins enables efficient sliced reads, as it allows join execution to happen in *stages*, few partitions at a time (*e.g.*, 1-10 out of 100). We have modified the job scheduler to schedule compute nodes in stages.

However, with intra-extent bucketing (§ 4.3), the bucketing within an extent is by a different dimension, whereas *sliced-read* reqiures the bucketing within an extent to be on the same dimension. Hence, in order to return slices, the storage node must locally re-order the records within the extent. As multiple compute nodes will read different slices of the same extent, a naive implementation that reads the entire extent, repartitions it and returns only the relevant slice, would result in excessive disk I/O (*e.g.*, 100x more disk reads for a slice-factor of 100). In-memory caching of the re-ordered extent data at the storage nodes can help, but incurs a memory cost proportional to the working set (the number of extents being actively processed).

To bridge this gap, the storage node performs co-ordinated lazy request scheduling, as it is aware of the pattern of requests during a join through a *sliced-read*. In particular, it knows that *slice-factor* compute nodes would be reading from the same extent, so it queues requests until a threshold number of requests (*e.g.*, 90%) for a particular extent arrives. It then reads the extent from disk, re-arranges the records by the right dimension and services the requests, and caches the chunks pertaining to the stragglers, *i.e.*, the remaining 10%. The cache usage reduces further by a factor of 10-100 with staging (above), *i.e.*, to less than 0.1%-1% of the input size. Thus, by exploiting compute-awareness to perform co-ordinated request scheduling and selective caching, *sliced-read* enables join execution without incurring any intermediate write to disk.

**Discussion:** Both localized shuffle and sliced reads for efficient joins require the cross-extent partitioning that super-extents provide, and do not work with a naive approach of simply bucketing within an extent, as all extents will have data from all partitions in that model. The small fan-in that super-extent partitioning enables, is crucial to the feasibility of co-ordinated scheduling.

## 7 Implementation

We have implemented *INSTalytics* in the codebase of a production analytics stack that handles exabytes of data. A key constraint is that to be practically deployable, the changes needed to be *surgical* and isolated without changing existing reasoning about recovery and failures. We describe in this section key aspects of the implementation.

### 7.1 System architecture

The file system in our analytics stack comprises of a Paxos-replicated *Master* that holds the metadata *in memory*, and Storage Nodes that store extents identified by *GUIDs*. The master maintains a mapping from a *file ID* to a list of extents in the file, and an offset-to-extent map. The master also tracks *extent metadata*, tracking their size, the list of replicas, *i.e.*, the storage nodes on which those instances are placed, *etc.*. A key constraint is to avoid increase in the in-memory metadata.

### 7.2 Changes to Filesystem Master

With logical replication, the extent is no longer a homogenous entity. For example, the sizes of the replicas in each dimension may be different due to skews in data distribution. To handle this, we hide the actual sizes of the extent instances from the master; the master continues to track extents and offsets in the physical space. The extent table at the master continues to track three instances per extent; we map them

to the same bucket index across the three logical dimensions (*e.g.*, instances of extent 0 of the stream would track the $0^{th}$ bucket of the three dimensions for first super-extent). We add a super-extent table to the stream metadata, tracking extent indexes that delimit super-extents; the total increase in metadata is $< 1\%$.

The extent placement logic is changed to handle super-extent aware fault isolation, and the recovery path is also changed minimally. Instead of asking a new storage node to copy extent data from a single source, the master sends a *list* of 300 sources for entire super-extent; the storage node talks to them to perform recovery.

### 7.3 Changes to Storage Nodes

The storage nodes handle most of the work for creation and recovery of logical extents. For each super-extent, the master sends a `logically_replicate` request to an orchestrator storage node, specifying the 300 physically replicated source instances, and 300 new destination storage nodes. The orchestrator sends a `create_logical_extent` request to each destination, on receipt of which the destination sends read requests to 100 source nodes to read one bucket's worth of data from each. It then assembles all that data into one extent by invoking the encoder of the format-specific adapter. The storage nodes that receive the bucketed read request invoke the *decoder* of the format-specific adapter to convert the extent into records, apply bucketing on the column values to return just the bucket the destination node is interested in; a decoded cache helps reuse this work across destinations. To amortize this cost across queries, logical replication is performed only on files that have a high read-write ratio. The storage nodes also track logical-extent-specific metadata in SSD that contain pointers to intra-extent sub-buckets, and additional rowID information to reconstruct data exactly as they were originally written. The recovery flow works similar to creating a logical extent, and handles parity recovery as well.

### 7.4 Changes to Compute Layers

Because of the adapter-based design at the storage nodes and the store's ability to reconstruct data with byte-level fidelity, compute layers continue to access storage through a block interface. The changes mostly have to deal with how the multiple dimensions are exposed to layers such as the query optimizer; the QO treats them as multiple clustered indexes. With our changes, the query optimizer can handle simple filters and joins by automatically picking the correct partition dimension; full integration into the query optimizer to handle complex queries is beyond the scope of this paper. The store provides a `filtered_read` API which returns only the subset of extents that match a given filter condition. Also, the store provides direct access to a specific dimension or specific bucket with file-name mangling (*e.g.*, filename[0]), for ease of integration. The client library of the store initiates *recovery-on-demand*; instead of trying a different replica for failover, it triggers an online recovery. As described in Section 4, this cost is identical to reading from another physical replica.

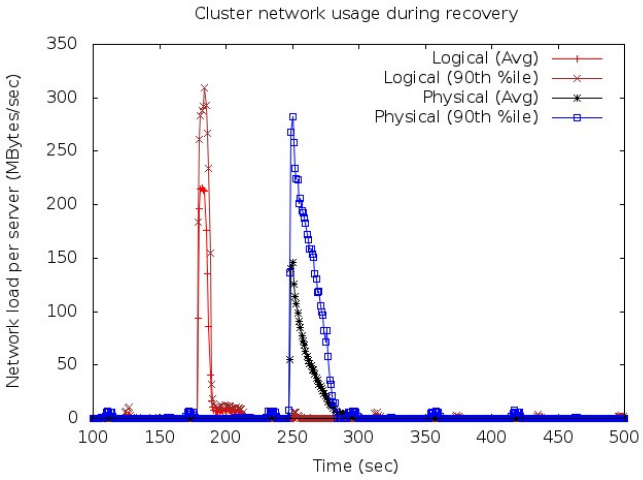


Figure 5: Cluster network load during recovery

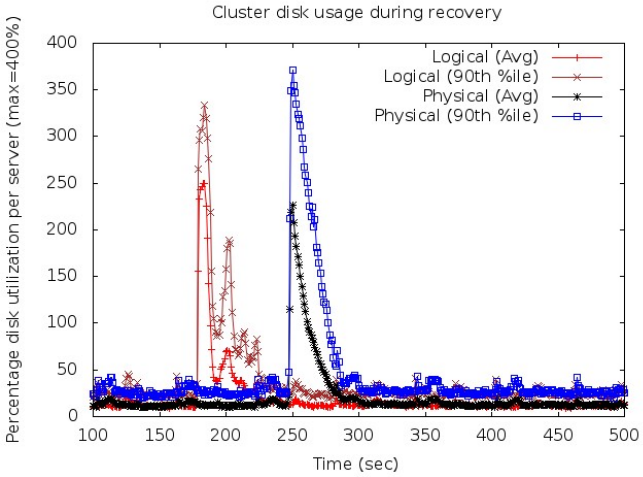


Figure 6: Cluster disk load during recovery

## 8 Evaluation

We evaluate *INSTalytics* in a cluster of 500 servers (20 racks of 25 servers each). Each server is a 2.4 GHz Xeon with 24 cores and 128 GB of RAM, 4 HDDs and 4 SSDs. 450 out of the 500 servers are configured as storage (and compute) nodes, and 5 as store master. We answer three questions in the evaluation:

- What is the recovery cost of the *INSTalytics* layout?
- What are the availability characteristics of *INSTalytics*?
- How much do benchmarks and real queries benefit ?

For our evaluation, unless otherwise stated, we use a configuration with 100 extents per superextent with an average extent size of 256MB.

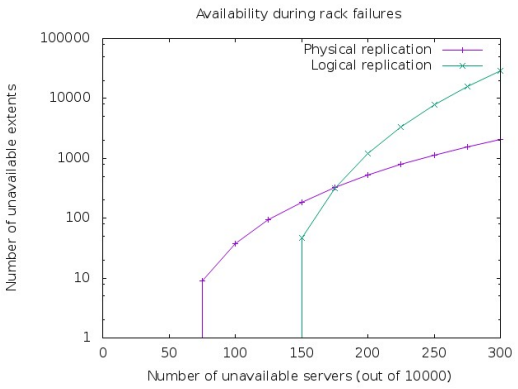


Figure 7: Storage availability during rack failures

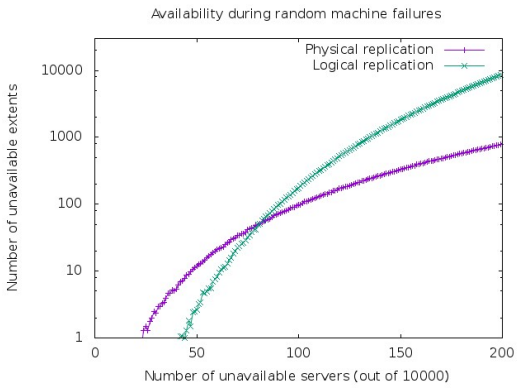


Figure 8: Storage availability during machine failures

## 8.1 Recovery performance

For this experiment, we take down an entire (random) rack of machines out of the 20 racks, thus triggering recovery of the extent replicas on those 25 machines. We then measure the load caused by recovery on the rest of the machines in the cluster. For a fair comparison, we turn off all throttling of recovery traffic in both physical and logical replication. During the physical/logical replication experiment, we ensure that all extents recovered belong to physically/logically replicated files respectively. The number of extents recovered is similar in the two experiments, about 7500 extents ($\tilde{1}$.5TB). We measure the network and disk utilization of all the live machines in the cluster, and plot average and $90^{th}$ percentiles. Although the physical and logical recovery are separate experiments, we overlay them in the same graph with offset timelines.

Figure 5 shows the outbound network traffic on all servers in the cluster during recovery. The logical recovery is more bursty because of its ability to read sub-buckets from 100 other extents. The more interesting graph is Figure 6 that shows the disk utilization of all other servers; because each server has 4 disks, the maximum disk utilization is 400%. As can be seen, the width of the two spikes are similar, which shows that recovery in both physical and logical complete with similar latency. The metric of disk utilization, together with the overall time for recovery, captures the actual work done by the disks; for instance, any variance in disk queue lengths caused by extra load on the disks due to logical recovery, would have manifested in a higher width of the spike in the graph. The spike in the physical experiment is slightly higher in the $90^{th}$ percentile, likely because it copies the entire extent from one source while logical replication is able to even out the load across several source extents. The key takeaway from the disk utilization graph is that the disk load caused by reading intra-extent chained buckets from 100 storage nodes, is as efficient as copying the entire extent from a single node with physical replication. The logical graph has a second smaller spike in utilization corresponding to the lag between reads and writes (all sub-buckets need to be read and assembled before the write can happen). For both disk and network, we summed up the aggregate across all servers during recovery and they are within 10% of physical recovery.

## 8.2 Availability

In this section, we compare the availability of logical and physical replication under two failure scenarios: isolated machine failures and co-ordinated rack-level failures. Because the size of our test cluster is too small to run these tests, we ran a simulation of a 10K machine cluster with our layout policy. Because of fault-isolated placement of buckets across dimensions, we tolerate up to 5 rack failures without losing any data (with parity, unavailability needs two failures in each dimension). Figure 7 shows the availability during pod failures. As can be seen, there is a crossover point until which logical replication with parity provides better availability than physical replication, and it gets worse after that. The cross-over point for isolated machine failures is shown in Figure 8 and occurs at 80 machines, *i.e.*, 0.8%. We also ran a simulation of double-parity layout; the cross-over points for isolated and correlated failures occur at 265 and 375 failures respectively.

To calibrate what realistic failure scenarios occur in practice, we analyzed the storage logs of a large cluster of tens of thousands of machines over a year to identify *dynamic failures*; failures which caused the master to trigger recovery (thus omitting long-dead machines etc.). We found that isolated machine failures are rare, typically affecting less than 0.2% of machines. There were 55 spikes of failures affecting more machines, but in all but 2 of those spikes, they were concentrated on one top-level failure domain, *i.e.*, a set of racks that share an aggregator switch. *INSTalytics* places the three dimensions of a file across multiple top-level failure domains, so is immune to these correlated failures, as it affects only one of three dimensions. The remaining 2 spikes had random failures of 0.8%, excluding failures from the dominant failure domain.

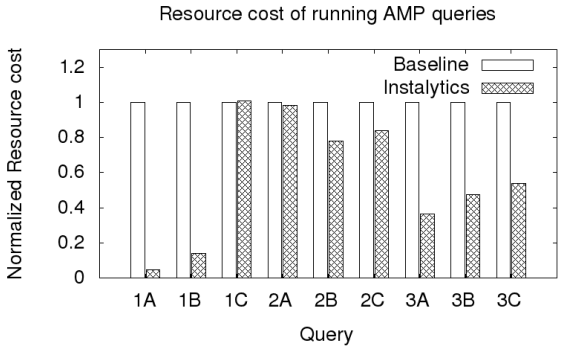


Figure 9: **Resource cost of AMP queries.**

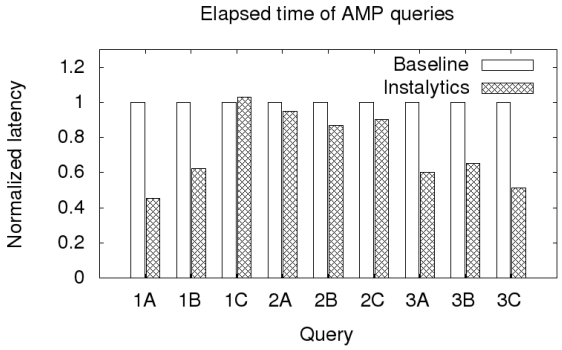


Figure 10: **Latency of AMP queries.**

## 8.3 Query performance

We evaluate the query performance benefits of *INSTalytics* on the AMP big-data benchmark [1, 16] and on a slice of queries from a production big data workload. We use two metrics: the query latency, and the "resource cost", *i.e.*, the number of machine hours spent on the query. All *sliced-read* joins use a request threshold of 90% and 10 stages. The baseline is a production analytics stack handling exabytes of data.

### 8.3.1 AMP Benchmark

The AMP benchmark has 4 query types: scan, aggregation, join, and external script query; we omit the last query, as our runtime does not have python support. As the emphasis is on large-scale workloads, we use a scale factor of 500 during data generation. We logically replicate the `uservisits` file (12TB) on 3 dimensons: url, visitDate, and IP and the `rankings` file (600GB) on 2: url and pagerank. Figures 9 and 10 show the benefit from *INSTalytics* in terms of cost and latency respectively. Queries 1A, 1B which perform heavy filtering on pagerank column of `rankings` benefit significantly from partitioning; the latency benefit is lower than the cost benefit because of the fixed startup latency. Query 3 does a join of the two files on the url column, after filtering `uservisits` on visitDate. Since both files are partitioned on url, 3C gets significant benefits while performing the join using sliced reads. 3A, 3B perform heavly filtering before the join and hence benefit from partitioning on the visitDate column. Queries 2A to 2C perform a group-by on a prefix of IP in `uservisits`, and get benefits of better local aggregation enabled by partitioning on IP. In summary, today one can pick one column out of three and get wins for a subset of queries, but lose benefits for other queries; *INSTalytics* simultaneously benefits all queries by supporting multiple dimensions.

Table 1 focuses on just the join within Q3 (excluding the aggregation that happens after the join). As can be seen, even the simple localized shuffle without sliced reads provides reasonable benefits. Further, we find that it reduces the amount of cross-rack network traffic by 93.4% compared to baseline. Sliced reads add to the benefit, providing nearly a 3x win for the join. To explore sensitivity to co-ordinated request scheduling, we show two configurations. In the "10% cache" configuration, the storage nodes wait for 90% of requests to arrive before serving the request; for the remaining 10% slices, the storage node caches the data. The "5%" configuration waits for 95% of requests. There is a tradeoff between cache usage and query performance; while the 5% configuration uses half the cache, it has a slightly higher query cost.

| Configuration | Cost (hrs) | Latency (mins) |
|---|---|---|
| Baseline | 125 | 11.8 |
| Localized Shuffle | 85 | 8.4 |
| Sliced Reads (10% cache) | 40.5 | 3.8 |
| Sliced Reads (5% cache) | 43 | 4.1 |

Table 1: **Performance of the join in Q3 of AMP benchmark**

### 8.3.2 Production queries

We also evaluate *INSTalytics* on a set of production queries from a slice of a telemetry analytics workload. The workload has 6 queries, all operating on an input file of size about 34 TB, sometimes joining with other files that are smaller. Table 2 shows the relative performance of *INSTalytics* on these queries. *INSTalytics* benefit queries Q1-Q4. Q5 and Q6 filter away very little data (<1%) and do not perform joins, so there are no gains to be had. Q1 and Q2 benefit from our join

| Description | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 |
|---|---|---|---|---|---|---|
| $Baseline_{cost}$ | 251 | 414 | 22 | 20 | 398 | 242 |
| $INSTalytics_{cost}$ | 59 | 206 | 0.3 | 1.1 | 403 | 239 |
| $Baseline_{latency}$ | 39 | 66 | 23 | 4 | 50 | 20 |
| $INSTalytics_{latency}$ | 7 | 21 | 1.4 | 2.3 | 51 | 20 |

Table 2: **Performance of production queries.** *Cost numbers are in compute hours, latency numbers are in minutes.*

improvements, they both join on one of the dimensions. Q1 performs a simple 2 way join followed by a group-by. We see a huge (>4x) improvement in both cost and latency as the join dominates the query performance (the output of the join is just a few GB). Q2 is more complex, it performs a 3 way join followed by multiple aggregations and produces multiple large outputs (3TB+). *INSTalytics* improves the join by about 3x and does as well as the baseline in the rest of the query. Q3 & Q4 benefit heavily from filters on the other two dimensions processing 34TB of data at interactive latencies.

As seen from our evaluation, simultaneous partitioning on multiple dimensions enables significant improvements in both cost and latency. As discussed in §3, 83% of large files in our production workload require only 4 partition dimensions for full query coverage, and hence can benefit from *INSTalytics*. The workload shown in §3 pertains to a shared cluster that runs a wide variety of queries from several diverse product groups across *Microsoft* so we believe the above finding applies broadly. As we saw in Figure 2, some files need more than 4 dimensions. Simply creating two different copies of the file would cover 8 dimensions, as each copy can use logical replication on a different set of 4 dimensions.

# 9 Related Work

**Co-design** The philosophy of cross-layer systems design for improved efficiency has been explored in data center networks [13], operating systems [3] and disk systems [23]. Like [13], *INSTalytics* exploits the scale and economics of cloud data centers to perform cross-layer co-design of big data analytics and distributed storage.

**Logical replication** The broad concept of logical redundancy has also been explored; the Pilot file system [20] employed logical redundancy of metadata to manage file system consistency, by making file pages self-describing. The technique that *INSTalytics* uses to make extents self-describing for format adapters is similar to the self-describing pages in Pilot. Fractured mirrors [18] leverages the two disks in a RAID-1 mirror to store one copy in row-major and the other in column-major order to improve query performance, but it does not handle recovery of one copy from the other. Another system that exploits the idea of logical replication to speed-up big-data analytics is HAIL [8]; HAIL is perhaps the closest related system to *INSTalytics*; it employs a simpler form of logical replication where they only reorder records within a single storage block; as detailed in Sections 4 and 6, such a layout provides only a fraction of the benefits that the super-extents based layout in *INSTalytics* provides (some benefit to filters but no benefit to joins). As we demonstrate in this paper *INSTalytics* achieves benefits for a wide class of queries without compromising on availability or recovery cost. Replex [25] is a multi-key value store for the OLTP scenario that piggybacks on replication to support multiple indexes for point reads with lower additional storage cost. The recovery cost problem is dealt with by introducing additional hybrid replicas. *INSTalytics* instead capitalizes on the bulk read nature of analytics queries and exploits intra-extent data layout to enable more efficient recovery, without the need for additional replicas. Further the authors do not discuss the availability implications of logical replication, which we comprehensively address in this paper. Erasure coding [19, 12] is a popular approach to achieve fault-tolerance with low storage-cost. However, the recovery cost with erasure coding is much higher than 3-way replication; the layout in *INSTalytics* achieves similar recovery cost as physical replication. Many performance sensitive analytics clusters including ours use 3-way replication.

**Data layout** In the big-data setting, the benefits of partitioning [30, 27, 24, 21] and co-location [7, 9] are well understood. *INSTalytics* enables partitioning and co-location on multiple dimensions without incurring a prohibitive cost. The techniques in *INSTalytics* are complementary to column-level partitioning techniques such as column stores [15]; in large data sets, one needs both column group-level filtering and row-level partitioning. Logical replication in *INSTalytics* can actually amplify the benefit of column groups by using different (heterogeneous) chocies of column groups in each logical replica within an intra-extent bucket, a focus of ongoing work.

# 10 Conclusion

The scale and cost of big data analytics, with exabytes of data on the cloud, makes it important from a systems viewpoint. A common approach to speed up big data analytics is to throw parallelism or use expensive hardware (*e.g.*, keep all data in RAM). *INSTalytics* provides a way to *simultaneously* both speed up analytics *and* drive down its cost significantly. *INSTalytics* is able to achieve these twin benefits by fundamentally reducing the actual work done to process queries, by adopting techniques such as logical replication and compute-aware co-ordinated request scheduling. The key enabler for these techniques is the co-design between the storage layer and the analytics engine. The tension in co-design is doing so in a way that only involves surgical changes to the interface, so that the system is pragmatic to build and maintain; with a real implementation in a production stack, we have shown its feasibility. We believe that a similar vertically-integrated approach can benefit other large-scale cloud applications.

# Acknowledgements

We thank our shepherd Nitin Agrawal and the anonymous reviewers for their valuable comments and suggestions. We thank Raghu Ramakrishnan, Solom Heddayya, Baskar Sridharan, and other members of the Azure Data Lake product team in Microsoft, for their valuable feedback and support, including providing access to a test cluster.

## References


[1] Amp big-data benchmark. In *https://amplab.cs.berkeley.edu/benchmark/*.

[2] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1383–1394.

[3] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and control in gray-box systems. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 43–56.

[4] BINDSCHAEDLER, L., MALICEVIC, J., SCHIPER, N., GOEL, A., AND ZWAENEPOEL, W. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 20:1–20:15.

[5] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on computers 44*, 2 (1995), 192–202.

[6] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[7] DITTRICH, J., QUIANÉ-RUIZ, J.-A., JINDAL, A., KARGIN, Y., SETTY, V., AND SCHAD, J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 515–529.

[8] DITTRICH, J., QUIANÉ-RUIZ, J.-A., RICHTER, S., SCHUH, S., JINDAL, A., AND SCHAD, J. Only aggressive elephants are fast elephants. *Proc. VLDB Endow. 5*, 11 (July 2012), 1591–1602.

[9] ELTABAKH, M. Y., TIAN, Y., ÖZCAN, F., GEMULLA, R., KRETTEK, A., AND MCPHERSON, J. Cohadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow. 4*, 9 (June 2011), 575–585.

[10] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[11] HSIAO, H.-I., AND DEWITT, D. J. Chained declustering: A new availability strategy for multiprocessor database machines. In *Data Engineering, 1990. Proceedings. Sixth International Conference on* (1990), IEEE, pp. 456–465.

[12] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 15–26.

[13] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 3–14.

[14] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[15] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases* (2010), pp. 330–339.

[16] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009), ACM, pp. 165–178.

[17] RAMAKRISHNAN, R., SRIDHARAN, B., DOUCEUR, J. R., KASTURI, P., KRISHNAMACHARI-SAMPATH, B., KRISHNAMOORTHY, K., LI, P., MANU, M., MICHAYLOV, S., RAMOS, R., SHARMAN, N., XU, Z., BARAKAT, Y., DOUGLAS, C., DRAVES, R., NAIDU, S. S., SHASTRY, S., SIKARIA, A., SUN, S., AND VENKATESAN, R. Azure data lake store: A hyperscale distributed file service for big data analytics. In *SIGMOD Conference* (2017).

[18] RAMAMURTHY, R., DEWITT, D. J., AND SU, Q. A case for fractured mirrors. *The VLDB Journal 12*, 2 (2003), 89–101.

[19] RASHMI, K., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 331–342.

[20] REDELL, D. D., DALAL, Y. K., HORSLEY, T. R., LAUER, H. C., LYNCH, W. C., MCJONES, P. R., MURRAY, H. G., AND PURCELL, S. C. Pilot: An operating system for a personal computer. *Communications of the ACM 23*, 2 (1980), 81–92.

[21] SHANBHAG, A., JINDAL, A., MADDEN, S., QUIANE, J., AND ELMORE, A. J. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 229–241.

[22] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[23] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems.

[24] SUN, L., KRISHNAN, S., XIN, R. S., AND FRANKLIN, M. J. A partitioning framework for aggressive data skipping. *Proc. VLDB Endow. 7*, 13 (Aug. 2014), 1617–1620.

[25] TAI, A., WEI, M., FREEDMAN, M. J., ABRAHAM, I., AND MALKHI, D. Replex: A scalable, highly available multi-index data store. In *USENIX Annual Technical Conference* (2016), pp. 337–350.

[26] THEKKATH, C. A., MANN, T., AND LEE, E. K. *Frangipani: A scalable distributed file system*, vol. 31. ACM, 1997.

[27] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1626–1629.

[28] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

[29] ZHANG, H., CHO, B., SEYFE, E., CHING, A., AND FREEDMAN, M. J. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 43:1–43:15.

[30] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-A., CHAIKEN, R., AND SHAKIB, D. Scope: Parallel databases meet mapreduce. *The VLDB Journal 21*, 5 (Oct. 2012), 611–636.

[31] ZHOU, J., LARSON, P. A., AND CHAIKEN, R. Incorporating partitioning and parallel plans into the scope optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (March 2010), pp. 1060–1071.