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

MUTHIAN SIVATHANU, MIDHUL VUPPALAPATI, BHARGAV S. GULAVANI, KAUSHIK RAJAN, and JYOTI LEEKA, Microsoft Research India
JAYASHREE MOHAN, University of Texas-Austin
PIYUS KEDIA, Indraprastha Institute of Information Technology Delhi

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 three-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 we 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.

CCS Concepts: • **Information systems → Storage replication**; **Cloud based storage**; *Database query processing*;

Additional Key Words and Phrases: Storage replication, data center storage, big data query processing

ACM Reference format:
Muthian Sivathanu, Midhul Vuppalapati, Bhargav S. Gulavani, Kaushik Rajan, Jyoti Leeka, Jayashree Mohan, and Piyus Kedia. 2020. INSTalytics: Cluster Filesystem Co-design for Big-data Analytics. *ACM Trans. Storage* 15, 4, Article 23 (January 2020), 30 pages.
https://doi.org/10.1145/3369738

Authors' addresses: M. Sivathanu, M. Vuppalapati, B. S. Gulavani, K. Rajan, and J. Leeka, Microsoft Research India, Vigyan No. 9, Lavelle Road, Bangalore, Karnataka, 560001; emails: muthian@microsoft.com, midhul.v@gmail.com, {bharg, krajan, jyleeka}@microsoft.com; J. Mohan, University of Texas-Austin, 2317 Speedway, Austin, Texas, 78712; email: jaya@cs.utexas.edu; P. Kedia, Indraprastha Institute of Information Technology Delhi, Okhla Industrial Estate, Phase III, Near Govind Puri Metro Station, New Delhi, Delhi, 110020; email: piyus@iiitd.ac.in.

1553-3077/2020/01-ART23 $15.00
https://doi.org/10.1145/3369738

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 [13, 21, 26] 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 [14]. 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 [2, 8, 35]. 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 [35], 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 [33].

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 article, 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 three-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 *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 up to an order of magnitude, in the efficiency of analytics processing.

The key contributions of the article 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 three-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 article is structured as follows. In Section 2, we provide a background of big-data analytics processing. In Section 3, we characterizate a production analytics workload. We present logical replication in Section 4, discuss its availability implications in Section 5, and describe optimizations for joins in Section 6. We present the implementation of *INSTalytics* in Section 7, and evaluate it in Section 8. We present related work in Section 9, and conclude in Section 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 a compute layer, such as MapReduce [8] or Spark [32], and a distributed file system, such as GFS [13] or HDFS [26]. 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 [13, 18, 30], 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 article). An extent is typically between 64 and 256 MB 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

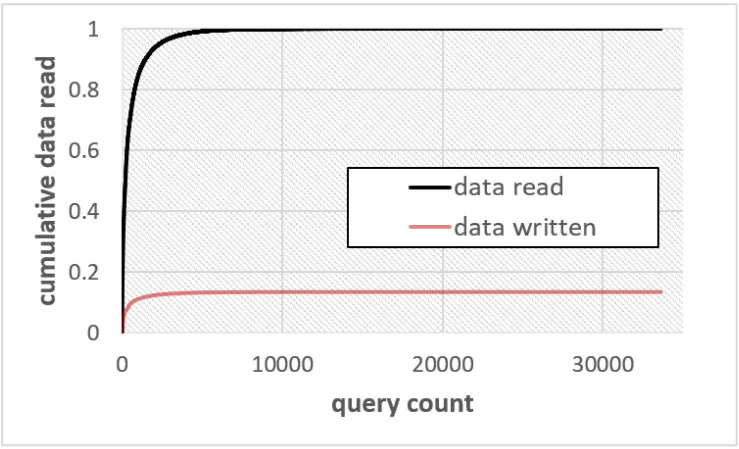


Fig. 1. Data filtering in production queries.

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, then 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, then 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 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 7× reduction in data sizes past the first stage. This reduction is due to both column-level and row-level filtering; while column-stores [19] 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

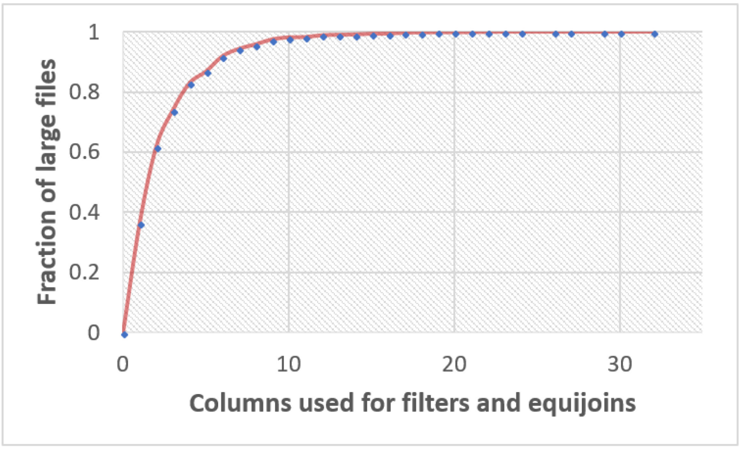


Fig. 2. Number of partition dimensions needed.

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 four partitioning dimensions, the coverage grows significantly to about 83%. Thus, for most files, having them partitioned in four dimensions would enable efficient execution for all queries on that file, as they would benefit from partitioning or co-location.

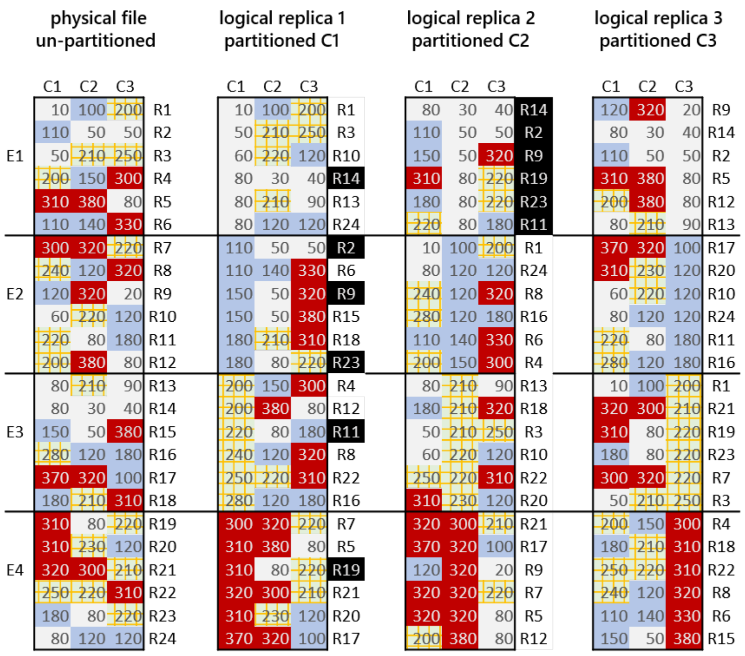
Supporting multiple dimensions: Today, the user can partition data across multiple dimensions by storing multiple copies. However this comes with a storage cost: to support four dimensions, the user incurs a 4× space overhead, and worse, a 4× cost in write bandwidth to keep the copies up to date. Interestingly, in our discussions with teams that perform big-data analytics within *Microsoft*, we found examples where large product groups actually maintain multiple (usually two) 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



	physical file un-partitioned				logical replica 1 partitioned C1				logical replica 2 partitioned C2				logical replica 3 partitioned C3			
	C1	C2	C3		C1	C2	C3		C1	C2	C3		C1	C2	C3	
	10	100	200	R1	10	100	200	R1	80	30	40	R14	120	320	20	R9
	110	50	50	R2	50	210	250	R3	110	50	50	R2	80	30	40	R14
E1	50	210	250	R3	60	220	120	R10	150	50	320	R9	110	50	50	R2
	200	150	300	R4	80	30	40	R14	310	80	220	R19	310	380	80	R5
	310	380	80	R5	80	210	90	R13	180	80	220	R23	200	380	80	R12
	110	140	330	R6	80	120	120	R24	220	80	180	R11	80	210	90	R13
	300	320	220	R7	110	50	50	R2	10	100	200	R1	370	320	100	R17
	240	120	320	R8	110	140	330	R6	80	120	120	R24	310	230	120	R20
E2	120	320	20	R9	150	50	320	R9	240	120	320	R8	60	220	120	R10
	60	220	120	R10	150	50	380	R15	280	120	180	R16	80	120	120	R24
	220	80	180	R11	180	210	310	R18	110	140	330	R6	220	80	180	R11
	200	380	80	R12	180	80	220	R23	200	150	300	R4	280	120	180	R16
	80	210	90	R13	200	150	300	R4	80	210	90	R13	10	100	200	R1
	80	30	40	R14	200	380	80	R12	180	210	320	R18	320	300	210	R21
E3	150	50	380	R15	220	80	180	R11	50	210	250	R3	310	80	220	R19
	280	120	180	R16	240	120	320	R8	60	220	120	R10	180	80	220	R23
	370	320	100	R17	250	220	310	R22	250	220	310	R22	300	320	220	R7
	180	210	310	R18	280	120	180	R16	310	230	120	R20	50	210	250	R3
	310	80	220	R19	300	320	220	R7	320	300	210	R21	200	150	300	R4
	310	230	120	R20	310	380	80	R5	370	320	100	R17	180	210	310	R18
E4	320	300	210	R21	310	80	220	R19	120	320	20	R9	250	220	310	R22
	250	220	310	R22	320	300	210	R21	320	320	220	R7	240	120	320	R8
	180	80	220	R23	310	230	120	R20	320	320	80	R5	110	140	330	R6
	80	120	120	R24	370	320	100	R17	200	380	80	R12	150	50	380	R15

Fig. 3. **Naive logical replication.** The figure shows the physical layout of the input file and the layout with naive logical replication. The file has four extents each consisting of six rows, three 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 to 99, E2 from 100 to 199 and so on). Recovering E1 from replica 2 requires reading all extents from replica 1.

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.

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 200 MB extent would require reading an entire say 10 TB file in another dimension, whereas with physical replication, only 200 MB 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

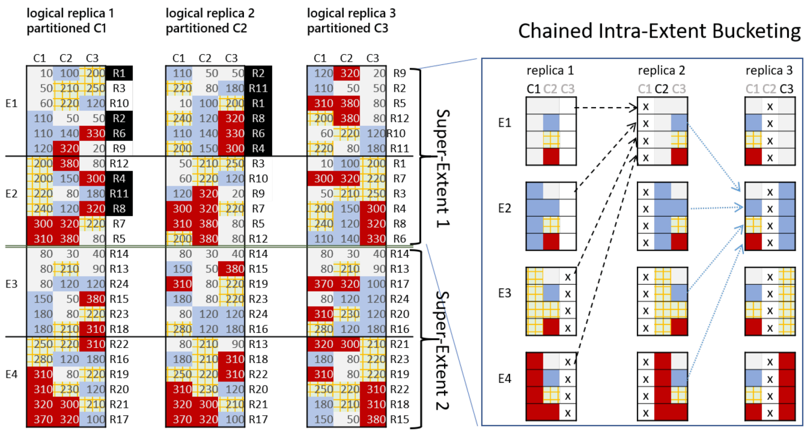

logical replica 1 partitioned C1
logical replica 2 partitioned C2
logical replica 3 partitioned C3
Super-Extent 1
Super-Extent 2
Chained Intra-Extent Bucketing
replica 1
replica 2
replica 3
C1 C2 C3
E1
E2
E3
E4

Fig. 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 E1 of replica 2 requires only reading slices from replica 1 whose C2 boxes are gray.

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 corresponding 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 to recover a 200 MB extent, the store has to read "only" 100 extents of that super-extent from another dimension. This is better than reading the entire say 10 TB file with the naive approach, but the cost of recovery is still significantly higher (100×) 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 1/1,000th 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 100×, whereas the file-level global partitioning could support 1,000 partitions and provide a larger benefit. However, the 100× 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, then 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 100× 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 replication. Given 100 extents per super-extent, the key-space of C_1 would be partitioned into 100 ranges, so the ith extent in every super-extent of dimension C_1 would contain records whose value for column C_1 fall in the ith 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 200 MB extent is now comprised of 100 buckets each roughly of size 2 MB. 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 first extent of replica 2, the store needs to read only the first bucket from extents E1 to E4 of replica 1, instead of reading the full content of those 4 extents. We refer to replica 1 as the friendly replica for recovery of extents in replica 2.

Thus, in a super-extent of 100 extents, instead of reading 100 × 200 MB to recover a 200 MB extent, we now only read 2 MB each from 100 other extents in a friendly replica, i.e., read 200 MB to recover a 200 MB extent, essentially the same cost as physical replication. By reading 100 chunks of size 2 MB each, we potentially increase the number of disk seeks in the cluster. However, given the 2 MB 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.

[1]This fourth dimension is not equally as powerful as the first three, because while it provides partition elimination for filters, it does not provide co-location for joins.

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

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 involves 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 co-ordination 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, and so on. 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 (three-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 three-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 is 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

The super-extent-based layout that we described in previous sections solves the recovery cost problem. However, a more subtle consequence of logical replication is that of *availability*. An extent is *unavailable* if ① the machine that contains it is down, and ② it cannot be recovered from other replicas (reads to the extent would then fail with a non-retryable error). With physical replication, this would be the case if and only if all three machines holding the three copies of the extent are down. Hence, if p is the independent probability that a specific machine in the cluster goes down within a fixed-time window, then the probability of unavailability in physical replication for a given extent is p^3. But with logical replication, an extent becomes unavailable if ① the machine with that extent is down and ① additionally *at least 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

$$\overbrace{p}^{①} \times \overbrace{(1-(1-p)^{100})(1-(1-p)^{100})}^{②}$$
$$= 10^4 . p^3 \pm O(p^4).$$

Assuming $p \ll 1$ and neglecting higher order terms in the Taylor-series expansion this approximates to $10^4 \cdot p^3$, i.e., the probability of unavailability is 10,000 times higher than that of physical replication. This is obviously problematic and needs to be dealt with. Note that in this reasoning, we only consider independent machine failures, as this is the worst-case. Correlated failures can be handled with fault-isolated placement as described later in this section.

5.1 Parity Extents

To handle this gap in availability, we introduce an additional level of redundancy in the layout. Within a super-extent replica, which comprises 100 extents, we add 1 *parity extent* with simple XOR-based parity for every group of 10 extents (to form a *parity-group* of 11 extents), i.e., a total of 10 parity extents per super-extent replica. Now, since each parity group can tolerate one failure, for an extent to be unavailable the following need to be true:

① The machine with that extent is down.
② There has to be at least one other failure in the extent's parity group.
③ At least one parity group in each of the other two dimensions should have ≥2 failures.

The expression for probability of unavailability is thus the conjunction of the probabilities of these three conditions being true:

$$\begin{aligned}
&\overbrace{p}^{①}\overbrace{\times\left(1-(1-p)^{10}\right)}^{②}\overbrace{\left(1-\left((1-p)^{11}+11\cdot p(1-p)^{10}\right)^{10}\right)^2}^{③}\\
&= p.\left(10\cdot p\pm O(p^2)\right)\left(10\cdot\binom{11}{2}\cdot p^2\pm O(p^3)\right)^2\\
&= p\times 10p\times 10.\binom{11}{2}.p^2\times 10.\binom{11}{2}.p^2\pm O(p^7)\\
&= 3.02\times 10^6.p^6\pm O(p^7)\\
&\approx 3.02\times 10^6.p^6,\ \text{if } p\ll 1.
\end{aligned}$$

When $3.02\times 10^6.p^6 < p^3$, the availability of this scheme would be better than that of physical replication. This is true, as long as ($p < 0.7\%$). 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 22 extents can tolerate two failures. The probability of unavailability then approximates to $p.\binom{21}{2}.p^2\times 5.\binom{22}{3}.p^3\times 5.\binom{22}{3}.p^3 = 12.45\times 10^9.p^9$ (assuming $p\ll 1$), 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 general expression for the probability of unavailability, given a super-extent size of s and assuming l parity extents are added to each group of k extents (assuming k divides s) such that each parity-group of $k+l$ extents can tolerate up to l failures, is as follows:

$$p\cdot\left(\sum_{i=l}^{k+l-1}\binom{k+l-1}{i}\cdot p^i(1-p)^{k+l-1-i}\right)\left(1-\left(\sum_{i=0}^{l}\binom{k+l}{i}\cdot p^i(1-p)^{k+l-i}\right)^{s/k}\right)^2.$$

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 (1 parity extent per 10 extents) is often sufficient, and hence this is what we have currently implemented in *INSTalytics*.

Recovering a parity extent requires 10× disk and network I/O compared to a regular extent, because it has to perform an XOR of 10 corresponding extents. As 10% of logically replicated extents are parity extents, this would double the average cost of recovering an extent ($90\%\times 1x + 10\%\times 10x\approx 2x$). We therefore store two physically replicated copies of each parity extent, so that during recovery, most of the failed parity extents can be recovered by simply copying data from the other replica, and we incur the 10× disk and network cost only for the tiny fraction of cases where both replicas of the parity extent are lost. This is a knob for the cluster administrator—whether to incur the additional 10% space cost, or the 2× average I/O cost during recovery; in our experience, recovery cost is more critical and hence the default is two-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 failure can take down all machines in that rack, distributed file systems perform fault-isolated placement. For example, the placement would ensure that the three replicas of an extent are placed in three different failure domains, to avoid simultaneously losing multiple copies of the extent. With logical replication, the records in a given extent are spread across multiple other extents in the other replicas of the super-extent, thus requiring a different strategy for fault-isolation. The fundamental low-level property we would like to ensure is that the *copies of any single record should be present in different failure-domains*. The way *INSTalytics* ensures this property is by reasoning about fault isolation at the super-extent level, because all replicas of a given super-extent contain exactly the same set of records. 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. It should be easy to see that doing this is sufficient to ensure the record-level fault isolation property.

Large-scale failures in production clusters are typically correlated in nature [12]. Fault-isolated placement enables *INSTalytics* to deal with these kinds of failures. For example, consider a data center that has three top-level failure domains (each containing 1/3rd of the machines). Even if all the machines in two failure domains (i.e., 66.6% of machines) are down, all extents would still remain available as one replica of each super-extent will still be intact.

5.3 Dealing with Query-time Failures and Stragglers

One of the key benefits of replication, is the ability to mitigate failures and stragglers that occur during query execution: the compute layer can simply fall back to another replica of the failed/slow extent, and hence incurs very little I/O cost. To achieve similar functionality in *INSTalytics*, we introduce the notion of *on-demand recovery*, where a compute task can recover an extent's data on the critical path during query execution. As described in Section 4, the I/O cost for doing this is identical to reading from another replica.

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 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 100×) 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 [33]. 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, then 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

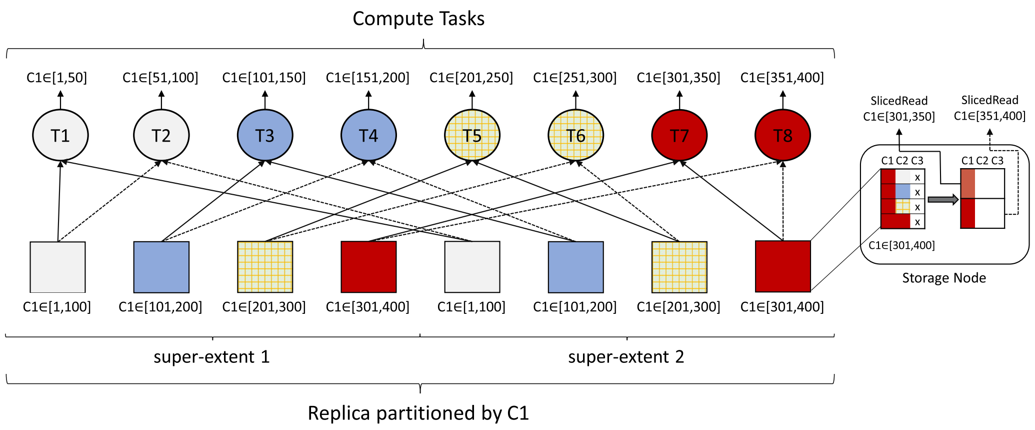


Fig. 5. **Using SlicedReads to obtain finer-grained partitions.** This figure shows how sliced-reads can be used to obtain finer-grained partitioning. There are 8 compute tasks, reading from a file with super-extent size 4. Below each extent is the range of C1 values that it contains. Above each compute task is the range of C1 values read by that task. Even though the file actually has only 4 partitions, using a slice-factor of 2 we are effectively able to get 8 partitions. Each compute task retrieves records pertaining to its C1 range, by performing sliced-reads on multiple extents.

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, then the join of each individual partition can happen in parallel in memory, avoiding this disk cost. However, as the super-extent-based layout limits 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 *SlicedRead*, which allows a client to read a small semantic-slice of an extent that contains records that belong to a sub-range of its overall key-range, further sub-dividing its key-range into *slice-factor* number of pieces. For instance, if an extent contains records with column C1 in range 0–10k, a *SlicedRead* with a *slice-factor* of 100 on this extent can ask for only records with column C1 in range 9k–9.1k (0–10k gets divided into 100 pieces of size 100 each: 0–100, 100–200, . . . 9k–9.1k, . . . 9.9k–10k). Using *SlicedRead*s the compute layer can effectively get the benefits of having a larger number of partitions. For example, with a super-extent size of 100 and a *slice-factor* of 100, one could get 10,000 partitions. For a 10 TB file, 10,000 partitions would mean each is of size 1 GB, and hence join on each pair of partitions can easily be processed in memory. For larger files, one can use larger *slice-factor* to obtain more partitions.

Figure 5 illustrates how this is possible. In this example, the input file is logically replicated with one of the replicas partitioned on C1. The super-extent size is 4, and there are a total of 8 extents, i.e., 2 super-extents. Since the super-extent size is 4, the layout gives us 4 partitions (*i*th extent of each super-extent forms partition *i*). Let us call these *Real Partitions*, as they are an artifact of the physical layout of data on disk. Using *SlicedRead*s the compute layer can effectively get $4 * 2 = 8$ partitions. Let us call these finer-grained partitions *Virtual Partitions*. The Figure shows 8 compute tasks, each reading 1 of these partitions. Each *Virtual Partition* is associated with one

Real Partition. *Virtual Partition*s are essentially sub-divisions of *Real Partition*s. In this example each *Real Partition* maps to 2 (*slice-factor*) *Virtual Partition*s. To read a *Virtual Partition*, under the hood, the compute task performs 1 *SlicedRead* on each extent of the corresponding *Real Partition*. Here, since each *RealPartition* consists of 2 extents, each compute task performs 2 (number of super-extents) *SlicedRead*s. For example Task T1 does *SlicedRead*s requesting records with C1 in 1–50 on extents 1 and 5, and with that it gets all records in the file with C1 in 1–50. Similarly each extent gets 2 (*slice-factor*) *SlicedRead* requests with disjoint key-ranges. For example, extent 1 gets one *SlicedRead* requesting range 1–50 from T1, and another *SlicedRead* requesting range 51–100 from T2.

It would be possible to serve *SlicedRead*s directly off disk, if the records inside an extent were sorted by the *SlicedRead* column. However, with chained intra-extent bucketing (Section 4.3), the ordering of records within an extent is by a different dimension. For example, as shown in the zoomed-in view of the last extent in Figure 5, the extent is internally bucketed by C2, while the *SlicedRead*s are on C1. Hence, 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, re-partitions it and returns only the relevant slice, would result in excessive disk I/O (e.g., 100× 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 *SlicedRead*. 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 slices pertaining to the stragglers, i.e., the remaining 10%. Thus, by exploiting compute-awareness to perform co-ordinated request scheduling and selective caching, *SlicedRead* enables join execution without incurring any intermediate I/O to disk.

Staged Execution. For efficient processing of *SlicedRead*s at the storage nodes, *INSTalytics* schedules join tasks at the compute layer in a staged fashion. Each stage consists of all tasks reading from a particular *RealPartition*. For example, in Figure 5, tasks T1 and T2 form stage 1, T3 and T4 form stage 2, and so on. This has two key benefits. First, it enforces implicit co-scheduling of all the tasks reading from the same *Real Partition*, avoiding possible deadlocks when there is fixed parallelism. Consider the example in Figure 5, with a parallelism (number of concurrent tasks allowed) of 2. Since there is no constraint on the order in which tasks can be scheduled, it is possible for instance that tasks T1 and T3 get scheduled first. This would result in a deadlock, since these tasks each read from a different *Real Partition*, and neither can make progress. Staging would avoid this from happening, ensuring that tasks T1 and T2 run first, followed by T3 and T4 and so on. The second benefit of staging is that it enables further reduction in cache usage. For example, if there are 10 stages and 10% of data is selectively-cached at storage nodes while processing *SlicedRead*s, then the cache usage would be bounded to 1/10th of 10% = 1% of the input file size.

Discussion

Both localized shuffle and sliced reads for efficient joins require the cross-extent partitioning that super-extents provide, and would not work with a naive approach of simply bucketing within an extent, as all extents will have data from all partitions in that model. With the super-extent-based layout, for *SlicedRead*s each compute task needs to touch only 1/100th of all extents in the file. This reduced fan-in is crucial to the feasibility of staged execution and co-ordinated scheduling.

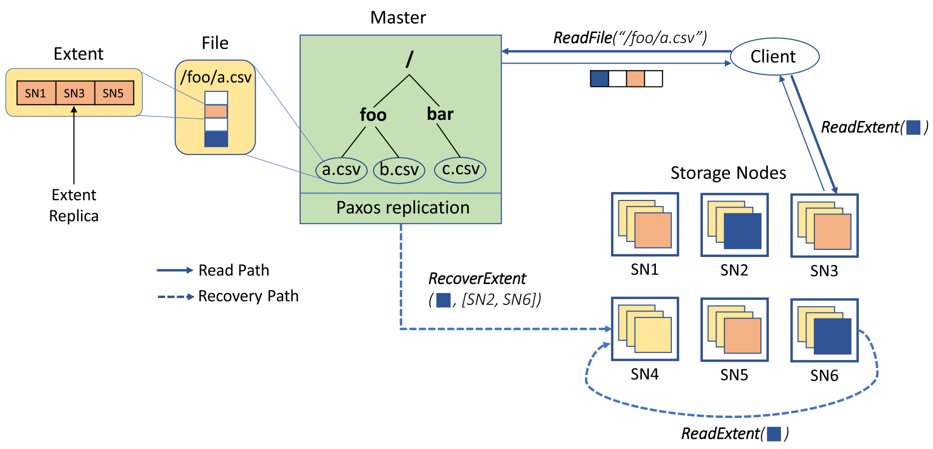


Fig. 6. **Architecture of existing filesystem.** The figure depicts the two main components of the filesystem: the Master and the Storage Nodes. Toward the left, it shows a zoomed-in view of the metadata that the Master stores for a file with four extents and that of one of its extents that contains three replicas. Two of the extents in the file are highlighted to show the locations of their replicas in the Storage Nodes. Towards the right, it shows the API calls made by the Client to read file data, and the operations involved in recovering an extent.

7 IMPLEMENTATION

We have implemented *INSTalytics* in the codebase of the analytics stack that is used in production internally at *Microsoft*. The stack has been in use since more than a decade and has significantly evolved over the years to meet the scale, performance, and reliability demands of first-party workloads. Today it is responsible for storing and processing several exabytes of data on a weekly basis. The architecture of various layers in the stack and their optimization have been the topic of numerous research papers [6, 7, 16, 21, 34]. We begin by describing the architecture of the existing cluster file system in our analytics stack. We then list some key constraints that we needed the implementation to conform to, to be pragmatic. Finally, we proceed to the actual details of the implementation.

7.1 Existing File System Architecture

The architecture of the file system in our analytics stack is very similar to that of a standard append-only distributed file system [13, 26], and is depicted in Figure 6. The two key components of the file system are the *Master* and the *Storage Node*s. The *Master* is a Paxos-based replicated service that is responsible for two things: First, it stores global file system level metadata in memory. This includes the directory hierarchy, files, extents, the location of extent replicas and any metadata associated with each of these entities. Second, it periodically synchronizes with the storage nodes, tracking their health and triggering recovery of extents when necessary. The *Storage Node*s are responsible for actually storing the the extents on disk. In addition, they also store some local per-extent metadata such as CRC checksums. A production cluster typically consists of several tens of thousands of *Storage Node*s.

The *Master* uses in-memory data structures to store file and extent level metadata. Files and extents are identified by globally unique IDs. The file data structure contains a list of extents that belong to the file. The extent data structure contains fields to track the length of the extent and the list of its replicas, i.e., the identifiers of the *Storage Node*s that have replicas of this extent.

7.1.1 Read Path. As illustrated in Figure 6, clients that want to scan data from a particular file first issue a *ReadFile* call to the *Master* to get the file's list of extents and the corresponding locations of their replicas. Then they issue a sequence of *ReadExtent* calls to the requisite *Storage Node*s to retrieve extent data.

7.1.2 Recovery Path. Since the *Master* periodically synchronizes with the *Storage Node*s it has a global view of the cluster state. When it detects that a particular *Storage Node* is unavailable, it triggers recovery operations for all the extents on that *Storage Node*. To recover an extent it picks a healthy *Storage Node* to house a new replica of the extent and sends it a *RecoverExtent* request along with information about the existing replicas of the extent (as illustrated in Figure 6). The *Storage Node* then reconstructs the extent locally by reading the extent data from a different healthy replica. The placement logic at the *Master* involved in picking the new *Storage Node* enforces failure isolation constraints of the new replica with respect to the existing replicas. At steady state, the *Master* tries to ensure each extent has exactly three healthy replicas, i.e., if the number of healthy replicas is less than three, then it triggers recovery, and if it is more than three, then it triggers deletion of the redundant replicas.

7.2 Constraints

For *INSTalytics* to be practically deployable, we had to adhere to certain constraints while making changes to the stack.

- The changes should be minimal and *surgical* to the extent possible.
- Since the *Master* stores all metadata in memory, any metadata change that materially increases the memory overhead is unacceptable. In particular, the number of extents that the *Master* tracks is several orders of magnitude higher than the the number of files. Consequently, we want to avoid introducing any additional extent-level metadata.
- Logically replicated files should be able to co-exist with physically replicated files. The changes should not introduce additional cost/overheads for operations/metadata pertaining to physically replicated files.
- Critical operations such as Read and Recovery for logically replicated data should not incur additional round-trips to *Master* or interactions between *Master* and *Storage Node*s (when compared to the same operations on physically replicated files).

In the following sections, we describe the key aspects of our implementation, while discussing the challenges introduced by the above constraints and the techniques we used to address them.

7.3 Metadata Changes

In this section, we describe the new metadata changes that we introduced in the *Master* and the *Storage Node* as part of the *INSTalytics* implementation. These changes form the basis for the various code paths that we describe in following sections.

7.3.1 Metadata in Master. With logical replication, the fundamental semantics of files and extents change: Files are now composed of an array of super-extents as opposed to a linear array of extents. Extents are no longer homogeneous entities: The data inside an extent is spread across multiple other extents in a different partition dimension, i.e., an extent no longer has exact "replicas." Instead of introducing new data structures to deal with the new semantics, the approach we take in our implementation is to piggyback on existing file/extent data structures, introducing additional fields in them only when absolutely necessary. This enables us to achieve our goals of minimal additional memory overhead and surgical code changes (Section 7.2).

We add a new flag to the file data structure to identify whether or not a given file is logically replicated. The file contains a linear array of extent as usual. Since the size of super-extents in a given file is fixed, in theory the super-extent boundaries can be implicit; i.e., extents 1–100 form super-extent 1; 101–200 form super-extent 2; and so on. The remainder of extents at the end, which are not sufficient to fill a super-extent, form the tail and remain physically replicated. However, we would like to support concatenation of files as a metadata-only operation (i.e., creating a new file that points to the extents in both the source files). When two files are concatenated, if the first file has a tail, that would lead to the presence of a hole/gap of physically replicated extents in the middle of the concatenated file. To deal with this more general scenario, we add a super-extent table to the file data structure, which tracks the offset of the starting extent of every super-extent. Note that the memory overhead of this table is very minimal (one integer per 100 extents).

The extent data structure continues to track three replicas as usual: We implicitly map these to the bucket index across all three partition dimensions. (e.g., replicas of extent 1 in a super-extent would track the first bucket of all three partition dimensions). Note that these are not really exact "replicas": This is just a trick we use to avoid additional metadata. The actual physical sizes of these replicas may be different due to skew in data distribution. To deal with this, we hide the actual sizes of the replicas from the *Master.* Another important piece of information that needs to be tracked is the identity of the extent replicas, i.e., which partition dimension each replica belongs to. We make this information implicit, by enforcing an order on the extent replica list, i.e., the first replica belongs to partition dimension 1, the second replica belongs to partition dimension 2 and so on. This works when the extent is in steady state, i.e., it has exactly 1 replica in each partition dimension. However, given the dynamics of the cluster (failure and recovery events), an extent could enter a state where it has 0 or more than 1 replicas of a given partition dimension. In this case it is not possible to track the identity of the replicas implicitly based on ordering. The naive approach to dealing with this issue is to add an additional field to the extent data structure to track the number of replicas in each partition dimension. As mentioned in Section 7.2, we cannot afford to do this, as it would increase the memory footprint at the master.

We solve this problem by introducing the notion of a *Conditional Header Replica.* In steady-state the extent has exactly three replicas and their partition dimensions are implicit. When the extents are not in steady-state, we add a dummy replica at the beginning of the replica list, which stores the counts of replicas per partition dimension. This dummy replica includes a magic number to enable detection of its presence. The logic of dealing with these two separate cases is abstracted away in the implementation of the extent data structure's accessor methods, hence obviating the need to modify other code paths that operate on this data structure. With this, only extents that are not in steady state bear the memory overhead of an additional replica in their list. Since, at any given point in time only a small fraction of all the extents are in non-steady state, the memory overhead of this approach is negligible.

7.3.2 Metadata in Storage Nodes. The efficient recovery scheme that we described in Section 4.3 requires reading buckets within extents. To enable this, the byte offsets of these buckets within each extent need to be stored, alongside the extent data in the *Storage Node*s. We do this by storing an additional *metadata extent* associated with every logically replicated extent. The data of this auxiliary extent contains the byte offsets of buckets in the actual extent. The ID of the metadata extent is computed by applying a deterministic hash function on the actual extent's ID. This enables mapping between the two without additional structures. The size of this metadata is negligible compared to the actual size of the extent. Also, the metadata extents are tracked locally by the *Storage Node*s and their lifetime is tied to their corresponding extents, so the *Master* does not need to explicitly track them. As a result, they do not lead to any additional memory overhead or complexity at the *Master.*

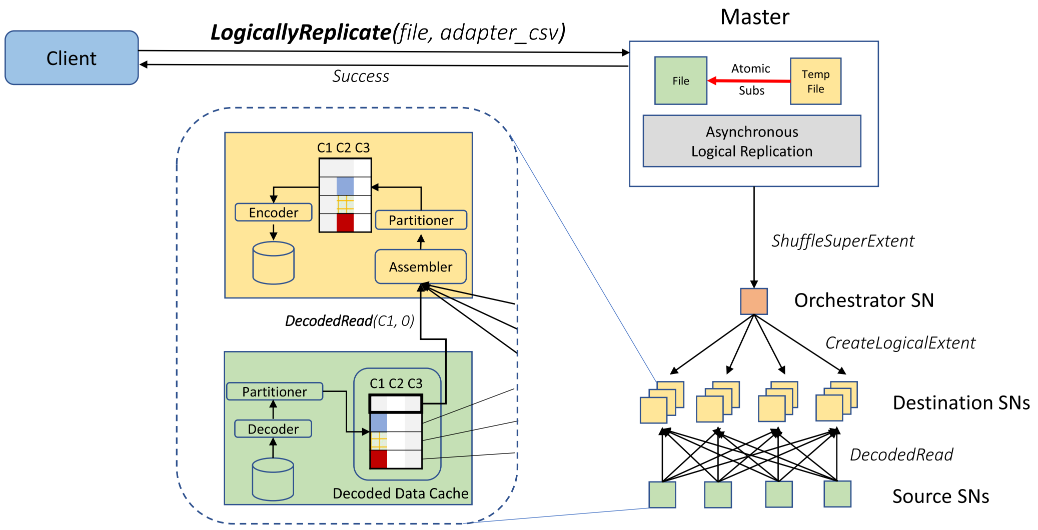


Fig. 7. **Creation of a logically replicated file.** The figure depicts the end-to-end flow of operations involved in the creation of a new logically replicated file. On the left is a zoomed-in view of the datapath inside Destination and Source Storage Nodes during a super-extent shuffle.

7.4 New APIs and Their Implementation

We now describe the new APIs that we introduced as part of the *INSTalytics* implementation. To illustrate how and when these APIs are invoked, we discuss three key code paths: the *Create*, *Recovery*, and *Read* paths.

7.4.1 Create Path. This is the path through which logically replicated files are introduced into the system. The flow of operations is as depicted in Figure 7. We add a new API *LogicallyReplicate* at the *Master*, that enables users to transform an existing physically replicated file into a logically replicated one. This way users can pick which subset of their files need logical replication, and these can co-exist with other physically replicated files in the system (which is a requirement as mentioned in Section 7.2). The *LogicallyReplicate* API takes as parameter, the URL of the target file, and other parameters relevant to logical replication such as the super-extent size, and the adapter that needs to be used depending on the file format.

Upon receiving the *LogicallyReplicate* request the *Master* performs two actions: First, it creates a new temporary file structure to serve as a placeholder for the new logically replicated file that is to be created. Then, it kicks off an asynchronous logical replication operation by enqueuing a job into the *Logical Replication Scheduler*. At this point the API call returns *Success* to the client and further processing happens asynchronously in the background. The client can poll the status of the original file to determine when the process is complete. The Scheduler breaks the logical replication job into tasks: Each task encapsulates the processing necessary to produce one super-extent in the file. Hence, the number of tasks is equal to the number of super-extents. For each task it picks an arbitrary *Storage Node* to serve as the *Orchestrator*. This *Orchestrator* is responsible for coordinating the process of shuffling data pertaining to a given super-extent. The amount of cluster resources used for logical replication can be controlled by tuning the number of concurrent tasks the Scheduler is allowed to run.

To kick off a task, the *Master* sends a *ShuffleSuperExtent* request to the *Orchestrator*. This request carries two key pieces of information: First, the locations of the *source extents*: extents in the original physically replicated file, which correspond to the super-extent that this *Orchestrator* is responsible for. Second, the identifiers of the *Destination Storage Node*s where the newly created extents should be placed. With a super-extent size of 100, the number of *source extent*s would be 100 (each having 3 replicas), and the number of *destination Storage Node*s would be 300. The goal is to read data from the *source extents*, shuffle it across the network and write it out to the *Destination Storage Node*s, such that the final layout of data is as described in Section 4.

In our implementation this shuffle is performed via a pull-based mechanism. The *Orchestrator* issues *CreateLogicalExtent* requests to each of the *destination Storage Node*s. Each of these requests carry the locations of all *source extent*s. In response to these requests, each *destination Storage Node* pulls the data that it requires from the *source extent*s. This is done by issuing *DecodedRead* requests to the *Storage Node*s containing the source extents. In addition to the extent id, the parameters to *DecodedRead* include the *column* to partition on and the *partition id*. For example, if replica 0 is to be partitioned on column C1, then the *destination Storage Node* responsible for replica 0 of the first extent in the super-extent would issue *DecodedRead(C1, 0)* on all the source extents. As a result it would end up fetching all the data corresponding to partition id 0 when partitioned by C1. Similary, the read for replica 0 of the next extent would issue *DecodedRead(C1, 1)* and so on.

The data-path for processing these *DecodedRead*s at the *Storage Node*s is shown in the zoomed-in view of Figure 7. First, the extent data is read from disk, then it is decoded using the decoder (which is supplied by the format adapter) to convert from byte-space to row-space. The rows are then partitioned based on the specified column, and rows corresponding to the required partition id are returned in response. In addition the decoded extent data is stored in the *Decoded Data Cache*, so that future *DecodedRead*s on the same extent can be served directly from cache. This ensures that each source extent is read from disk only once during the shuffle process. The zoomed-in view of Figure 7 also shows the data path inside the *destination Storage Node*s. Once all the *DecodedRead*s return, their responses are assembled, following which the data is partitioned by a different column and passed through the adapter-supplied encoder before finally being written to disk. Note that the partitioning by different column is needed to support intra-extent chained buckets described in Section 4.3. After the new extent's data has been successfully written to disk, the *Storage Node* returns success on the *CreateLogicalExtent* call made by the *Orchestrator*.

Once all *CreateLogicalExtent* calls at the *Orchestrator* return, it returns success on the *ShuffleSuperExtent* call from the *Master*, along with metadata corresponding to the newly created extents. The master then appends extent structures for these new extents to the temporary file. Given that multiple super-extent shuffles are processed concurrently, they could complete out of order. To deal with this the *Master* buffers shuffle completions and processes them in the right order. Once all of the super-extent shuffles are complete and all of the new extents are appended to the temporary file, the original file is atomically substituted with the temporary file. At this point the creation process is complete, and the original file has been successfully transformed into a logically replicated one.

Failure Handling. During the creation process, failures can happen and dealing with these is essential to ensure that the process eventually completes. Any of the components involved in the process can fail at any point in time: This includes *source Storage Node*s, *destination Storage Node*s, *Orchestrator*, and even the *Master*. Our implementation is equipped to handle all of these failure cases. When a Source *Storage Node* fails during a *DecodedRead*, the Destination *Storage Node*s simply fall back to a different replica of the source extent. To deal with *destination Storage Node* failure, the *Master* provides the *Orchestrator* with a pool of redundant *Storage Node*s on which *CreateLogicalExtent* calls can be retried upon failure. If the Orchestrator fails, then the partially complete shuffle is discarded and a fresh one is initiated by the *Master* on a new Orchestrator.

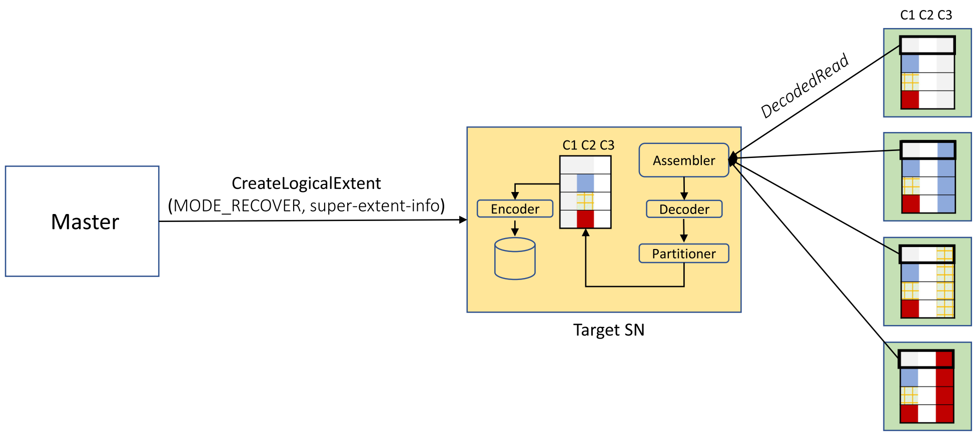


Fig. 8. **Recovery of a logical extent.** This figure depicts the flow of operations involved in the recovery of an extent in a logically replicated file. The Master invokes a request on the target Storage Node to trigger recovery. Inside the target SN, the figure shows the datapath involved in processing the recovery request by reading intra-buckets from the friendly replica's extents.

This could lead to work being wasted, but since there is only 1 Orchestrator per super-extent the probability of Orchestrator failure is low and this is not a significant problem. Finally, when the *Master* fails, it falls back to a secondary. Since the in-memory state of the *Master* is Paxos replicated, the secondary will have a copy of the temporary file structure, and it can resume processing from whichever point the old *Master* stopped.

7.4.2 Recovery Path. The high-level flow of operations (as shown in Figure 8) in the recovery path for extents in logically replicated files is very similar to that for those in physically replicated files (Section 7.1.2): the master sends a request to a *Storage Node* to trigger the recovery of an individual extent, and the *Storage Node* takes care of the processing necessary to produce a local copy of the required extent. This satisfies our constraint of no additional interactions between *Master* and *Storage Node*s (Section 7.2). The key differences lie in how the *Master* decides *when to trigger recovery for an extent, information passed in the recovery request* and *how the* Storage Node*s process recovery requests.*

The condition in the *Master* for triggering extent recovery today is: *extent has less than three healthy replicas.* This condition is not sufficient for extents in logically replicated files. For example, such an extent could have two healthy replicas in partition dimension 0, 1 in partition dimension 1, and 0 in partition dimension 2. Even though it has three healthy replicas, we still need to trigger recovery as one of the partition dimensions has 0 healthy replicas. Essentially, the recovery trigger condition needs to be aware of partition dimensions. For extents in logically replicated files, we change the condition to: *at least one partition dimension has no healthy replicas.*

Once the *Master* decides to recover an extent, it picks a target *Storage Node* to house a new replica of the extent and sends it a *CreateLogicalExtent* request with mode set to *MODE_RECOVER*. While making this choice, it enforces super-extent level fault-isolation (Section 5.2). In contrast to physical extent recovery, where only the list of existing replicas of the extent to recover are passed in the request, here we pass information about all extent replicas in the corresponding super-extent to which the extent belongs.

The data-path involved in processing the recovery request at the *Storage Node* is similar to that of the *Destination Storage Node*s during the create path. The target *Storage Node* issues *DecodedRead* requests to all of the extents in the friendly replica.[2] Since these extents are already internally bucketed by the required column, the *DecodedRead* can be serviced by simply fetching the required bucket from disk. Once all *DecodedReads* have returned, the target *Storage Node* assembles the received data, decodes it with the adapter-supplied decoder, partitions it by a different column (requirement of the layout in Section 4.3) and passes it through the adapter-supplied encoder before finally writing to disk. Note that since the create and recovery data-paths are similar, a lot of code in the *Storage Node* is shared between the two. This greatly minimizes the amount of code changes.

Failure Handling. If any of the extents in the friendly replica becomes unavailable during the process, then the target *Storage Node* performs *Recursive Recovery*, i.e., it first recovers the lost extent from its friendly replica and then proceeds with the actual recovery. The implementation also contains logic to use/recover parity extents when necessary.

7.4.3 Read Path. Extent Reads. Because of the adapter-based design at the *Storage Node*s, clients can continue to read extents through the normal block-based interface, i.e., this path does not require any modifications. We only change the client library of the file system to initiate *on-demand recovery* (Section 5.3) upon failure while reading an extent.

Filtered Reads. To accrue the benefits offered by the layout, filter queries running in the compute layer need the ability to read only a specific subset of the extents in a file based on the filter condition. We provide *filtered read* functionality at the storage layer for this purpose; the end-to-end read path consists of making a *filtered read* on the file to get information of only the required subset of extents, followed by a sequence of *ReadExtent* calls. This is analogous to the read path for physically replicated files (Section 7.1.1), with a *filtered read* instead of *ReadFile*. Hence, this satisfies our constraint of no additional round-trips to *Master* (Section 7.2).

Instead of introducing a new API for this purpose, we implement this functionality through the notion of *Logical Views* at the *Master*. For example if a client is interested in only the fifth partition of replica 0 of a logically replicated file *foo.csv*, it can simply reference *foo.csv:logicalview[0][5]*. This is essentially a "view" of the file that contains only the extents belonging to partition 5 in replica 0. For all practical purposes these views are equivalent to normal files from the perspective of the compute layer. Under the hood, within the storage layer however, these are not really separate files: this logic is implemented by simply modifying the *ReadFile* request processing at the *Master* to detect the presence of these special suffixes (*:logicalview[][]*) at the end of the filename, and return only the required subset of extents.

Sliced Reads. For *Sliced Reads*, as described in Section 6.2, we introduce a new API at the *Storage Node*s. The processing necessary for this API is very similar to that of *DecodedRead*s at source *Storage Node*s in the create path. Hence, for the implementation we re-use most of this functionality, leading to minimal additional code changes. For example the *Decoded Data Cache* is used to store data that needs to be selectively cached during sliced reads.

7.5 Changes to Compute Layers

The changes mostly have to deal with how the multiple dimensions are exposed to layers such as the query optimizer; the optimizer 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 article.

[2]Recall that with chained intra-extent bucketing, extents for a replica on one dimension can be efficiently recovered from a friendly replica that is intra-extent bucketed by that dimension (Section 4.3).

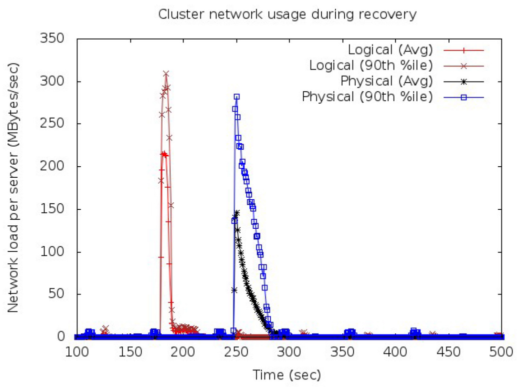


Fig. 9. Cluster network load during recovery.

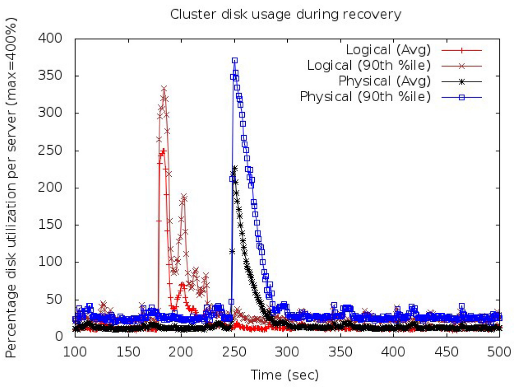


Fig. 10. Cluster disk load during recovery.

In addition, we had to make minimal changes to the *JobManager* (based on Dryad graph execution engine), to enable staged execution of join compute tasks for sliced reads. We achieve this adding null edges between nodes of the job graph to express artificial dependencies between tasks.

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. Out of the 500 servers, 450 are configured as storage (and compute) nodes, and 5 as store master. We answer three questions in the evaluation:

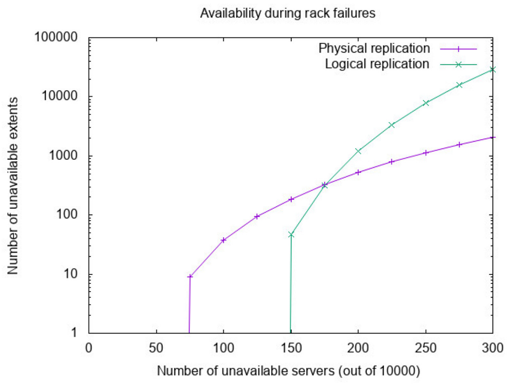
- What is the recovery cost of the *INSTalytics* layout?
- What are the availability characteristics of *INSTalytics*?
- How much do benchmarks and real queries benefit?

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

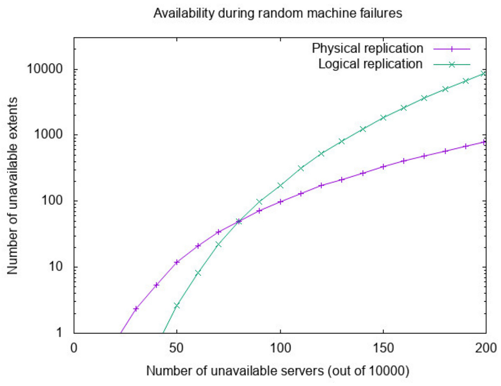
8.1 Recovery Performance

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 7,500 extents (~1.5 TB). We measure the network and disk utilization of all the live machines in the cluster, and plot average and 90th percentiles. Although the physical and logical recovery are separate experiments, we overlay them in the same graph with offset timelines.

Figure 9 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 10 that shows the disk utilization of all other servers; because each server has four 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



(a) Rack failures



(b) Machine failures

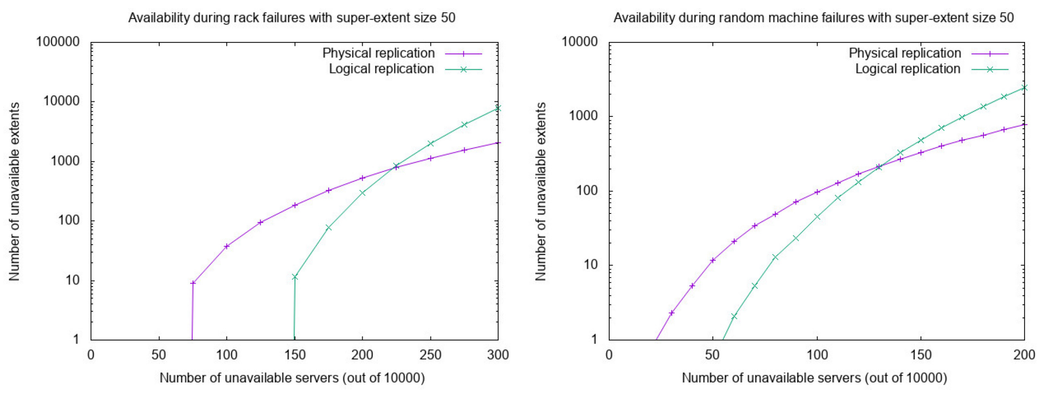
Fig. 11. Availability with single parity.

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 90th 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

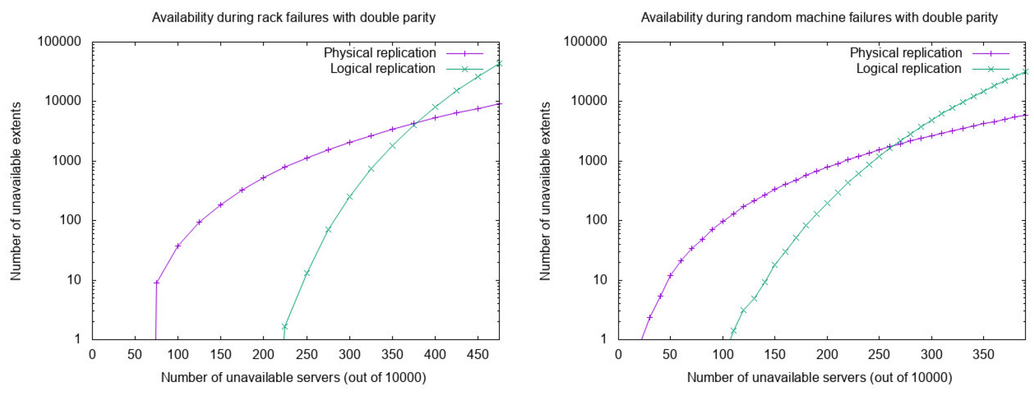
In this section, we compare the availability of logical and physical replication under two failure scenarios: isolated machine failures and co-ordinated rack-level failures. Because the size of our test cluster is too small to run these tests, we ran a simulation of a 10K machine cluster with our layout policy. Because of fault-isolated placement of buckets across dimensions, we tolerate up to five rack failures without losing any data (with parity, unavailability needs two failures in each dimension). Figure 11(a) 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 11(b) and occurs at 80 machines, i.e., 0.8%. We also ran simulations for the configuration where the super-extents contain 50 extents (instead of 100 extents in the default case), with 1 parity extent per 10 extents. Finally, we also evaluate availability for the double-parity configuration (super-extents of size 100, and 2 parity extents per 20 extents); The results are shown in Figures 12 and 13, 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 that 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



(a) Rack failures (b) Machine failures

Fig. 12. Availability with super-extent size 50.



(a) Rack failures (b) Machine failures

Fig. 13. Availability with double parity.

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.

8.3 Query Performance

We evaluate the query performance benefits of *INSTalytics* on the AMP big-data benchmark [1, 20] 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 four 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 (12 TB) on three dimensons: url, visitDate, and IP and the `rankings` file (600 GB) on 2: url and pagerank. Figures 14 and 15 show the benefit from *INSTalytics* in terms of cost and latency, respectively.

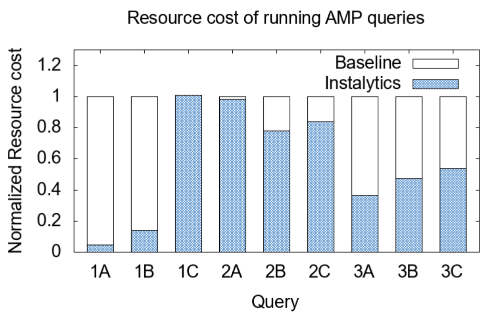


Fig. 14. Resource cost of AMP queries.

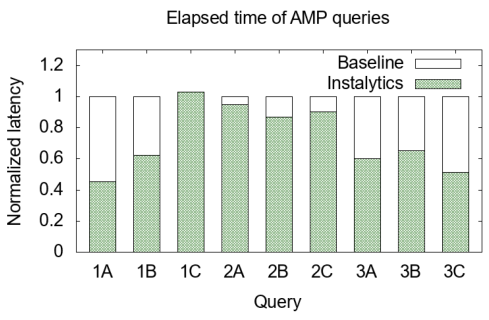


Fig. 15. Latency of AMP queries.

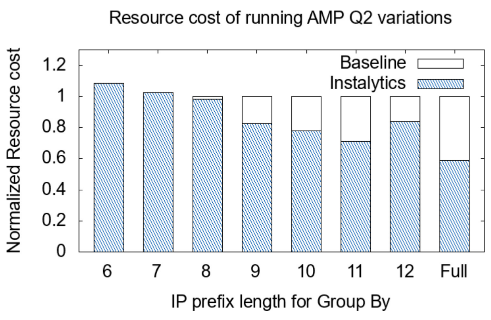


Fig. 16. Resource cost of AMP Q2 variations.

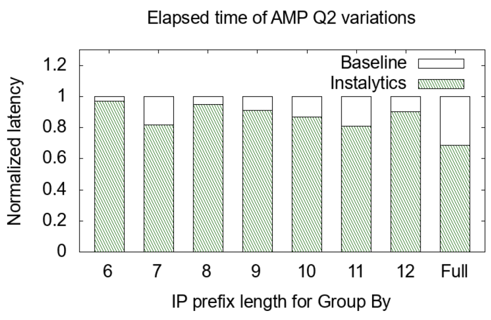


Fig. 17. Latency of AMP Q2 variations.

Query 1. This query filters `rankings` on the pagerank column and Queries 1A–1C are variations with different selectivities. Queries 1A and 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 1C scans almost all records in the file and hence does not see any benefits.

Query 2. This query performs a group-by of `uservisits` on a prefix of the IP column. Queries 2A–2C use different prefix lengths. Group-by is performed using a map-reduce style shuffle. Local aggregation is first done on the mapper side and global aggregation is done on the reducer side. Local aggregation reduces the amount of data that needs to be shuffled. The benefits for Queries 2A–2C are primarily due to better local aggregation enabled by partitioning on IP column (we use range partitioning so that IPs with similar prefixes end up together).

To better understand the group-by benefits, we ran more variations of query 2 with different IP prefix lengths. The cost and latency benefits are shown in Figures 16 and 17, respectively. Interestingly, we observe a sweet spot at around a prefix length of 11, where we get maximum benefits of local aggregation. This is because at lower prefix lengths, the baseline itself gets good local aggregation and partitioning does not add much value. At higher prefix lengths, we do not get much local aggregation. Group-by on the full IP (last bar in figures) is a special case. Since there is a replica partitioned on this column, we can use sliced-reads and do away with the need for on-disk shuffle.

Query 3. This query performs a join of the two files on the url column followed by a group-by on IP column. Before the join it filters `uservisits` on visitDate. Queries 3A–3C are variations with different selectivities for the filter. Since both files have replicas partitioned on url, Query 3C gets significant benefits while performing the join using sliced reads. Queries 3A and 3B perform

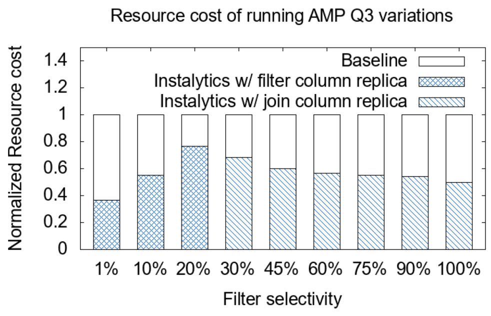


Fig. 18. Resource cost of AMP Q3 variations.

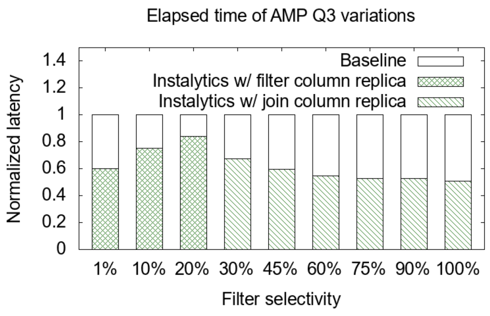


Fig. 19. Latency of AMP Q3 variations.

Table 1. Performance of the Join in Q3 of AMP Benchmark

Configuration	Cost (h)	Latency (min)
Baseline	125	11.8
Localized Shuffle	85	8.4
Sliced Reads (10% cache)	40.5	3.8
Sliced Reads (5% cache)	43	4.1

heavy filtering before the join and hence benefit from the replica of `uservisits` partitioned on the visitDate column.

This brings to light an interesting observation: depending on the selectivity of the filter before the join one could choose to exploit either the replica of `uservisits` partitioned on the filter column or the one partitioned on the join column. (Note that these choices are mutually exclusive). To further study this, we ran more variations of Query 3 with nine different selectivities. Figures 18 and 19 show the resulting benefits in cost and latency, respectively. Below a selectivity of 20%, we observe that it is better to exploit partitioning on the filter-column to get optimal benefits. Beyond 20% selectivity, it is better to exploit partitioning on the join column and use sliced-reads to perform the join.

Discussion. In summary, today one can partition the two files on only one dimension each. This would lead to benefits for only a subset of queries, but lose benefits for other queries; *INSTalytics* enables simultaneously benefits on all queries by supporting partitioning on multiple dimensions. Even within a single query (e.g., Query 3), different variations could entail partitioning by different columns for optimal benefits. In these kinds of scenarios *INSTalytics* can deliver significant benefits.

Join Microbenchmarks. Table 1 focuses on just the join within Q3 (excluding the filter before and group-by after the join). As can be seen, even the simple Localized Shuffle (Section 6.1) 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 (Section 6.2) add to the benefit, providing nearly a 3× win for the join. To explore sensitivity to co-ordinated lazy request scheduling during sliced-reads (Section 6.2), 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.

Table 2. Performance of Production Queries

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

Cost numbers are in compute hours, latency numbers are in minutes.

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 six 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 improvements, they both join on one of the dimensions. Q1 performs a simple two-way join followed by a group-by. We see a huge (>4×) 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 three-way join followed by multiple aggregations and produces multiple large outputs (3TB+). *INSTalytics* improves the join by about 3× and does as well as the baseline in the rest of the query. Q3 and Q4, because of their selectivities, benefit heavily from filters on the other two dimensions, processing 34 TB 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 Section 3, 83% of large files in our production workload require only four partition dimensions for full query coverage, and hence can benefit from *INSTalytics*. The workload shown in Section 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 four dimensions. Simply creating two different copies of the file would cover eight dimensions, as each copy can use logical replication on a different set of four dimensions.

9 RELATED WORK

Co-design The philosophy of cross-layer systems design for improved efficiency has been explored in data center networks [17], operating systems [3], and disk systems [27]. Like Reference [17], *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 [24] 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 [22] 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 [10]; 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-extent-based layout in *INSTalytics* provides (some benefit to filters but no benefit to joins). As we demonstrate in this article, *INSTalytics* achieves benefits for a wide class of queries without compromising on availability or recovery cost. Replex [29] 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 article. Erasure coding [15, 23] is a popular approach to achieve fault-tolerance with low storage-cost. However, the recovery cost with erasure coding is much higher than three-way replication; the layout in *INSTalytics* achieves similar recovery cost as physical replication. Many performance sensitive analytics clusters including ours use three-way replication.

Data layout In the big-data setting, the benefits of partitioning [25, 28, 31, 34] and co-location [9, 11] 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 [19]; 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.

ACKNOWLEDGMENTS

We thank our FAST 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] AMPLab. [n.d.]. AMP big-data benchmark. Retrieved from https://amplab.cs.berkeley.edu/benchmark/.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, New York, NY, 1383–1394. DOI: https://doi.org/10.1145/2723372.2742797

[3] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. 2001. Information and control in gray-box systems. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 43–56.

[4] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the 13th EuroSys Conference (EuroSys'18)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3190508.3190532

[5] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Comput.* 44, 2 (1995), 192–202.

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

[7] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya et al. 2019. Hydra: A federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 177–191.

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. DOI: https://doi.org/10.1145/1327452.1327492

[9] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 515–529. DOI: https://doi.org/10.14778/1920841.1920908

[10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. 2012. Only aggressive elephants are fast elephants. *Proc. VLDB Endow.* 5, 11 (July 2012), 1591–1602. DOI: https://doi.org/10.14778/2350229.2350272

[11] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. CoHadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.* 4, 9 (June 2011), 575–585. DOI: https://doi.org/10.14778/2002938.2002943

[12] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX. https://www.usenix.org/conference/osdi10/availability-globally-distributed-storage-systems.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 29–43. DOI: https://doi.org/10.1145/945445.945450

[14] H.-I. Hsiao and David J. DeWitt. 1990. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of the 6th International Conference on Data Engineering*. IEEE, 456–465.

[15] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX-ATC'12)*. USENIX, Boston, MA, 15–26. Retrieved from https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang.

[16] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 59–72.

[17] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM Conference on SIGCOMM (SIGCOMM'13)*. ACM, New York, NY, 3–14. DOI: https://doi.org/10.1145/2486001.2486019

[18] Edward K. Lee and Chandramohan A. Thekkath. 1996. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices*, Vol. 31. ACM, 84–92.

[19] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive analysis of web-scale datasets. In *Proceedings of the 36th International Conference on Very Large Data Bases*. 330–339. Retrieved from http://www.vldb2010.org/accept.htm.

[20] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of data*. ACM, 165–178.

[21] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. 2017. Azure data lake store: A hyperscale distributed file service for big-data analytics. In *Proceedings of the SIGMOD Conference*.

[22] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. 2003. A case for fractured mirrors. *VLDB J.* 12, 2 (2003), 89–101.

[23] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 331–342. DOI: https://doi.org/10.1145/2619239.2626325

[24] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. 1980. Pilot: An operating system for a personal computer. *Commun. ACM* 23, 2 (1980), 81–92.

[25] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. 2017. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, 229–241. DOI: https://doi.org/10.1145/3127479.3131613

[26] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE Computer Society, Washington, DC, 1–10. DOI: https://doi.org/10.1109/MSST.2010.5496972

[27] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2003. Semantically smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, 73–88. http://dl.acm.org/citation.cfm?id=1090694.1090702.

[28] Liwen Sun, Sanjay Krishnan, Reynold S. Xin, and Michael J. Franklin. 2014. A partitioning framework for aggressive data skipping. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1617–1620. DOI: https://doi.org/10.14778/2733004.2733044

[29] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. 2016. Replex: A scalable, highly available multi-index data store. In *Proceedings of the USENIX Annual Technical Conference*. 337–350.

[30] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. 1997. *Frangipani: A Scalable Distributed File System*. Vol. 31. ACM.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1626–1629. DOI: https://doi.org/10.14778/1687553.1687609

[32] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache spark: A unified engine for big-data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. DOI: https://doi.org/10.1145/2934664

[33] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. 2018. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3190508.3190534

[34] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel databases meet mapreduce. *VLDB J.* 21, 5 (Oct. 2012), 611–636. DOI: https://doi.org/10.1007/s00778-012-0280-z

[35] J. Zhou, P. A. Larson, and R. Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the IEEE 26th International Conference on Data Engineering (ICDE'10)*. 1060–1071. DOI: https://doi.org/10.1109/ICDE.2010.5447802

Received July 2019; revised October 2019; accepted October 2019