BullFrog: Online Schema Evolution via Lazy Evaluation

Souvik Bhattacherjee*† Gang Liao* Michael Hicks Daniel J. Abadi
ServiceNow† University of Maryland, College Park
souvik.bhattacherjee@servicenow.com {gangliao, mwh, abadi}@cs.umd.edu

ABSTRACT

BullFrog is a relational DBMS that supports single-step schema migrations — even those that are backwards incompatible — without downtime, and without need for advanced warning. When a schema migration is submitted, BullFrog initiates a logical switch to the new schema, but physically migrates affected data lazily, as it is accessed by incoming transactions. BullFrog's internal concurrency control algorithms and data structures enable concurrent processing of schema migration operations with post-migration transactions, while ensuring exactly-once migration of all old data into the physical layout required by the new schema. BullFrog is implemented as an open source extension to PostgreSQL. Experiments using this prototype over a TPC-C based workload (supplemented to include schema migrations) show that BullFrog can achieve zero-downtime migration to non-trivial new schemas with near-invisible impact on transaction throughput and latency.

ACM Reference Format:
Souvik Bhattacherjee, Gang Liao, Mike Hicks and Daniel J. Abadi. 2021. BullFrog: Online Schema Evolution via Lazy Evaluation. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452842

1 INTRODUCTION

Continuous Deployment (CD), an aspect of DevOps [13], is the increasingly popular practice of frequent, automated deployment of software changes [16, 36], with some practitioners deploying multiple changes per day [34]. To realize the benefits of CD, it must be straightforward to deploy updates to both front-end code and the database, *even when the database's schema has changed.* Unfortunately, this is where current practices run into difficulty. Savor et al. [34] stated in their retrospective on CD practices at OANDA, "database schema changes always were always ad hoc and full of fear." Claps et al. [16] and Shahin et al. [36] confirmed via surveys of more than 100 experts and practitioners that database schema changes are particularly challenging to handle. Nonetheless, there is evidence that schema changes are also frequent: Qiu et al. [30] examined changes in releases to a dozen open-source applications and found that schema changes occurred roughly once per week, on average. In our own examination of the development history of 20 open source Ruby on Rails applications found on GitHub (including the Sharetribe and GitLab repositories studied by Bailis et al [11]) we found 1,611 schema changes, of which approximately 20% required significant physical data movement.

Application developers should be free to change the code and database schema as they see fit, without concern for the complexities of deploying those changes later. For example, they should be able to delete or add columns or constraints, join tables, split tables, etc. according to their needs. To deploy an update, the developer defines an *evolution transaction* that is used to migrate the existing data to the new schema. The CD system uses that transaction to update the current database, and then deploy the updated front-end instances. If downtime is not a concern, then the simplest way to do this is to shut down all of the application instances, migrate the data, and then restart with the new instances. Of course, downtime frequently *is* a concern, and with multiple updates happening per day, the simple shutdown-and-restart approach is unacceptable.

State-of-the-art approaches to schema migration without downtime use a **multi-step** approach. In such approaches, the database system migrates a copy of the data to the new schema in the background, before the front-end instances can switch over to it. During this migration window, writes to the old schema must be propagated to the new copy, typically via triggers or log shipping of updates [1–5, 26, 32, 40]. Aside from the obvious side effect that such approaches approximately double the resource requirements of the system, the requirement to **delay** the front-end application migration until the database is ready is fundamentally antithetical to the spirit of the continuous deployment movement.

To avoid such delays, new-version transactions can be rewritten so they work on the old schema while the data is migrated; rewriting can occur in the other direction too, to allow both application versions to coexist [10, 17, 18, 20, 31, 32]. Unfortunately this limits schema evolution to backwards compatible migrations, since transactions must be rewritable/processable over all active schema versions. For example, if an application evolves such that data must be inserted that violates previously defined integrity constraints, the constraints cannot simply be dropped in the new schema, because doing so would be backwards-incompatible: this data cannot be inserted into the old schema. To cope, developers may be forced to employ non-ideal structures and/or temporary tables and front-end code, thereby accruing technical debt in applications and *decay* in databases [38], while adding complexity to the update process (see the OANDA quote above). All of these problems have pushed application developers toward "NoSQL" databases that avoid predefined schemas entirely, and never reject writes that use a different schema than has been previously used.

† Work performed during postdoc at University of Maryland, College Park.
* The first two authors contributed equally to this research.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452842

In this paper we propose a mechanism for schema evolution, that we call BullFrog[1], that avoids the delayed deployment of the new schema required in the multi-step approaches, while also avoiding the migration restrictions and database decay of the backwards compatible approaches. BullFrog deploys **arbitrary** schema changes immediately, in a **single step**, in conjunction with CD-based updates to a web-based or mobile service. Rather than introduce downtime by halting existing service while the evolution transaction takes place, BullFrog *logically* converts the database to use the new schema immediately without any *physical* changes of the stored data. Updated front-end instances may now submit transactions using the new schema. Doing so prompts BullFrog to migrate any relevant tuples from the old schema's tables to the new/updated ones before processing the transaction; i.e., tuples are migrated *lazily*, as needed. For backward-compatible schema changes, BullFrog permits old-version front-end instances to be updated gradually; for incompatible changes, front-end instances are updated as a *big flip* [15]; the latter can be done by simply restarting them (e.g., when they submit an incompatible query [33]), or using a more sophisticated dynamic software updating scheme [25, 28, 29].

BullFrog uses a light-weight concurrency control mechanism that ensures exactly-once migration of data under contention. Shared data structures synchronize record migration states and allow database system workers that are processing separate transactions to cooperate in parallel without missing or duplicating tuples, and without landing in stuck states due to aborts or cyclic dependencies.

We have implemented a prototype of BullFrog as an extension to PostgreSQL. We use this prototype to evaluate BullFrog by measuring its performance on variations of the standard TPC-C benchmark that include schema migration transactions. We find that under realistic deployment scenarios, BullFrog is able to support complex schema migrations in a near invisible fashion, with no downtime, no observable effects on system throughput, and limited latency increases. BullFrog is thus the first system, to the best of our knowledge, to support single-step, non-backwards compatible schema migrations without downtime.

In summary, this paper makes the following contributions:

- A proposed system design (BullFrog) that uses *lazy schema migration* to implement single-step, on-line schema evolution.
- Algorithms and data structures that achieve efficient, exactly-once physical migration of data under contention.
- An implementation of BullFrog in PostgreSQL. The code is currently available at https://github.com/DSLAM-UMD/BullFrog.
- An extension of the TPC-C benchmark that includes non-trivial and non-backwards compatible schema migrations.

2 REQUEST-DRIVEN LAZY MIGRATION

2.1 Basic approach

A schema migration request is submitted to BullFrog as one or more DDL statements. These statements may create new tables or modify or delete existing tables. Data from existing tables may be specified to initialize or update new or modified tables. The new schema becomes immediately active as soon as it is processed by BullFrog. For non backwards-compatible "big flip" migrations, the old schema becomes inactive, and all subsequent requests that access it are rejected. For requests over the new schema, BullFrog identifies tuples in the old tables that are potentially relevant, physically migrates them to the new/modified tables, and then processes the original request on the new/modified tables.

Consider a hypothetical airline flight application with an original schema containing two tables:

```
CREATE TABLE FLIGHTS (FLIGHTID CHAR(6) PRIMARY KEY, SOURCE
  CHAR(3), DEST CHAR(3), AIRLINEID CHAR(2), DEPARTURE_TIME
  TIMESTAMP, ARRIVAL_TIME TIMESTAMP, CAPACITY INT);
CREATE TABLE FLEWON (FLIGHTID CHAR(6), FLIGHTDATE DATE,
  PASSENGER_COUNT INT CHECK (PASSENGER_COUNT > 0));
```

The FLIGHTS table contains general information about active flight routes, and the FLEWON table contains daily flight statistics.

At some point the application developer makes some schema changes: (i) rename FLEWON to FLEWONINFO; (ii) add a derived attribute, EMPTY_SEATS; (iii) add attributes ACTUAL_DEPARTURE_TIME and ACTUAL_ARRIVAL_TIME to track the delay incurred by each flight; and (iv) drop constraint (PASSENGER_COUNT > 0) to allow for the airline to take packages rather than passengers during a pandemic. Change (iv) is backward-incompatible.

This change is expressed with the following migration DDL:

```
CREATE TABLE FLEWONINFO AS (
  SELECT F.FLIGHTID AS FID, FLIGHTDATE, PASSENGER_COUNT,
         (CAPACITY - PASSENGER_COUNT) AS EMPTY_SEATS,
         DEPARTURE_TIME AS EXPECTED_DEPARTURE_TIME,
         NULL AS ACTUAL_DEPARTURE_TIME,
         ARRIVAL_TIME AS EXPECTED_ARRIVAL_TIME,
         NULL AS ACTUAL_ARRIVAL_TIME
  FROM   FLIGHTS F, FLEWON FI
  WHERE  F.FLIGHTID = FI.FLIGHTID);
```

Once the migration has been submitted, BullFrog creates a new transaction that creates new, empty tables corresponding to the tables that are created or modified in the migration request, along with a temporary VIEW (that will only be used during the migration process) that contains the contents of the migration request:

```
CREATE VIEW FLEWONINFO_VIEW AS (
  SELECT F.FLIGHTID AS FID, FLIGHTDATE, PASSENGER_COUNT,
         ...); -- exactly matches FLEWONINFO def above
```

BullFrog also creates data structures (described in Section 3) to control concurrent migration processes and track migration status.

When the database system receives a client request that references any table that was added/modified, BullFrog first migrates any relevant data from the old tables, and then processes the original request over the new ones. To do this, it uses filtering statements in the client request, typically located in the WHERE clause of SELECT, UPDATE, and DELETE statements, to limit the scope of the lazy migration. BullFrog attempts to convert these filters over the new schema into filters over the old schema that match as few tuples as possible while still yielding the set needed to fully process the client request.

As an example, consider the following client request.

```
SELECT * FROM FLEWONINFO WHERE FID = 'AA101'
    AND EXTRACT(DAY FROM FLIGHTDATE) = 9;
```

[1]Bullfrogs do not sleep, much like our zero-downtime migration system.

The FID = 'AA101' predicate will be converted into FLIGHTID = 'AA101' over the FLIGHTS and FLEWON tables, while the predicate EXTRACT(DAY FROM FLIGHTDATE) = 9 will be applied on the FLEWON table. Only those tuples from these old tables that return true for these converted predicates need to be migrated; all other tuples can be ignored and migrated at a later time.

Although for this example it was straightforward to convert the predicates over the new schema into predicates over the old schema, in some cases such a conversion is non-trivial or impossible. In the worst case, all tuples in the old schema must be deemed potentially relevant (see Section 2.4).

BullFrog uses the VIEW that it created during the migration initialization step discussed above to leverage existing capabilities in database systems to move filters across schemas. Most database systems implement *view expansion* to rewrite requests over a view into requests over the original tables. BullFrog accesses the query plan that was generated after view expansion and query optimization, and extracts any filtering statements over the old schema.

For the example client request, the output from PostgreSQL EXPLAIN (which shows its query plan) is shown below:

```
                              QUERY PLAN
-----------------------------------------------------------------
Nested Loop (cost=4.34..14.04 rows=1 width=99)
->  Seq Scan on flights f (cost=0.00..4.50 rows=1 width=27)
    Filter: (flightid = 'AA101'::bpchar)
->  Bitmap Heap Scan on flewon fi
    (cost=4.34..9.52 rows=1 width=15)
    Recheck Cond: (flightid = 'AA101'::bpchar)
    Filter: (date_part('day'::text, (flightdate)::timestamp
    without time zone) = '9'::double precision)
    ->  Bitmap Index Scan on flewon_flightid_idx
    (cost=0.00..4.34 rows=9 width=0)
            Index Cond: (flightid = 'AA101'::bpchar)
```

The predicates over the view have been converted into predicates over the original tables; we see the predicate FLIGHTID = 'AA101' on both FLIGHTS and FLEWON table (line 5 and 13 in the query plan) and the predicate EXTRACT(DAY FROM FLIGHTDATE) = 9 on the FLEWON table (line 9-10 in the query plan). BullFrog inserts these predicates into a version of the original schema migration DDL, except that the CREATE TABLE statement in the query is substituted with an INSERT INTO statement, as shown below.

```
INSERT INTO FLEWONINFO (
  FID, FLIGHTDATE, PASSENGER_COUNT, EMPTY_SEATS,
  EXPECTED_DEPARTURE_TIME, ACTUAL_DEPARTURE_TIME,
  EXPECTED_ARRIVAL_TIME, ACTUAL_ARRIVAL_TIME)
(SELECT F.FLIGHTID, FLIGHTDATE, PASSENGER_COUNT,
       (CAPACITY - PASSENGER_COUNT),
       DEPARTURE_TIME, NULL, ARRIVAL_TIME, NULL
 FROM  FLIGHTS F, FLEWON FI
 WHERE F.FLIGHTID = FI.FLIGHTID
 AND   F.FLIGHTID = 'AA101' AND FI.FLIGHTID = 'AA101'
 AND   EXTRACT(DAY FROM FLIGHTDATE) = 9);
```

BullFrog implements DELETE and UPDATE statements by rewriting them into SELECT statements on the old schema to migrate relevant tuples first, and then processing the original request on the new schema. This limits BullFrog's reliance on views to read-only queries for which view expansion is trivial (via nesting SQL statements in the FROM clause), and avoids the well-known problem of performing updates through views.

INSERT commands generally can be performed over the new schema without requiring any prior migration unless there are integrity constraints defined on the new schema. Such constraints may expand the set of potentially relevant data beyond the data specified by the client request. For example, if a uniqueness constraint is defined on any column of the new table, then any INSERT commands over the new schema (or updates to the unique attribute) must first migrate records that have potentially conflicting values so that the constraint can be properly checked over the new schema.

2.2 Background migrations

If parts of the input tables are never deemed relevant for client requests, a purely lazy system will never migrate them. To ensure that all data is eventually migrated, BullFrog initiates background migration threads that slowly inject simulated client requests that cumulatively cover the entirety of the old tables. When these threads finish, the migration is complete and the old schema can be deleted.

2.3 Consistency

Unlike many of the state-of-the-art schema migration approaches discussed in Section 1, BullFrog does not maintain replicas as part of its approach. Logically, a given tuple starts in the old schema and eventually migrates to the new schema, but never exists in both schemas simultaneously. Once it is migrated, although a stale physical copy may remain in the old schema, the migration status of that tuple prevents it from subsequently being accessed. Thus, there is no concern for replica consistency in BullFrog[2].

With regard to ACID consistency, BullFrog does not restrict what constraints may be declared on the old schema or new schema. However, it does not automatically generate integrity constraints based on the integrity constraints that existed in the old schema. Rather, the schema migration DDL must explicitly (re)declare any integrity constraints that must be enforced on the new schema. We will analyze the performance impact of needing to migrate coarser units of data in order to check integrity constraints in Section 4.5.

2.4 Limitations

BullFrog supports any legal SQL that can appear in DDL statements in the database system, including any legally defined integrity constraint over the new schema. However, some types of migrations reduce BullFrog's effectiveness.

First, although BullFrog utilizes views in a read-only fashion, (thereby avoiding the view update problem), any limitations in the view support of the underlying database system is passed through to BullFrog. One situation where this limitation is manifested is when the migration is expressed using a user-defined function (UDF) instead of standard SQL. Many database systems support the incorporation of calls to UDFs within views, but treat the UDF as a black box during query planning. Any filtering conditions that appear within the UDF code will be invisible to BullFrog and therefore will not be helpful to limit the scope of the lazy migration.

[2]Such as the notion of consistency used by the CAP [14] and PACELC [9] theorems.

Second, integrity constraints added during migration may cause arbitrary data to be dropped. For example, if a uniqueness constraint is added to a table with duplicates, most existing systems will return an error immediately upon the ALTER TABLE command that attempted to add the constraint. However, a pure lazy migration approach would prevent the system from becoming aware of the problem until after the new schema is already live. Therefore, BullFrog must either perform a synchronous check upon receiving a potentially problematic migration command so that it can return an error in advance, or otherwise proceed with the pure lazy approach and give a warning that some records may fail to migrate.

3 LAZY MIGRATION, CONCURRENTLY

BullFrog must support concurrent client requests that may access overlapping sets of data. Care must be taken to avoid migrating the same tuple more than once, or deleting it from the old tables prematurely. BullFrog addresses these problems by using custom data structures and mechanisms to track the status of a tuple as it is migrated. At a high level, the technique involves locking ranges of data in the old tables to ensure that once one worker begins to migrate the data, no other concurrent worker can attempt to migrate it unless the first worker fails. Care is taken to handle situations in which there is not a one-to-one mapping of tuples under the old schema to tuples under the new schema. Furthermore, efficient data structures are used to track the status of these locks and the history of previously migrated data. In sum, BullFrog's design allows conflicting migration efforts to continuously and efficiently make progress migrating non-conflicting records, and avoid duplicating work for conflicting records.

3.1 Migration categories

A schema migration may involve one or more migration statements. Each migration statement may involve one or more tables from the old schema ("input tables") and generate one or more tables in the new schema ("output tables").[3] For each input table in a migration statement, BullFrog classifies it into four broad categories that dictate how its tuples will be locked and tracked during migration.

One-to-one (1:1) migration. In a 1:1 migration, each tuple in an input table has at most one corresponding tuple in the output schema (across all the output tables). Examples of 1:1 migrations include adding one or more columns to a table, dropping one or more columns from a table, changing the data type of a column, adding constraints to a table (which may cause the output table to be a subset of the tuples in the input table), or joining a table to another one using one of its foreign keys, i.e., a foreign-key, primary-key (FK-PK) join. For 1:1 migrations, BullFrog uses a bitmap to track migration and lock status. There are two bits per tuple in the input table: one bit corresponding to that tuple's migration status, and the other corresponding to its lock status. Bitmaps provide a favorable space-time trade-off, since they are effective at exploiting bit-level parallelism in hardware and introduce limited overhead. Tuple-level granularity on migration and lock status gives BullFrog the flexibility to migrate exactly those tuples that it has determined to be potentially relevant to a particular client request without having to drag along unnecessary data during the migration process.

[3]We ignore migrations that involve zero input or output tables since they are trivial.

Algorithm 1: Per-transaction migration loop.

```
do
    WIP, SKIP = empty list
    Start transaction
    Scan via client request predicates, for each tuple, T do
        canMigrate = Call Algorithm 2 or 3 for T
        if canMigrate == true then Include T in migration.
    End transaction
    for each tuple or group, G in WIP do
        Update G's status to migrated (not in-progress)
while SKIP is not empty
Run client request on new schema
```

However, BullFrog also provides the capability to track migration and lock status at less granular levels (e.g. at a page level).

One-to-many (1:n) migration. In a 1:n migration, each tuple in the input table may (but does not necessarily) produce more than one tuple in the output schema. One example of such a migration is where an input table is split into multiple output tables, with a single input tuple generating a tuple in each of the output tables. Other examples include the primary key side of a FK-PK join and either side of a many-to-many join. 1:n migrations work similarly to 1:1 migrations in that a bitmap is used to track migration and lock status. The only additional detail is that the migration bit for a tuple in the input table cannot be set (indicating that it has been migrated) until all of its dependent tuples in the output schema have been generated.

Many-to-one (n:1) migration. In a n:1 migration, a group of tuples from the same input table combine to generate a single tuple in the output schema. An example of an n:1 migration is where an output table is formed by performing a group-by aggregation. For these migrations, BullFrog tracks migration and lock status at the group level, and uses a hash table instead of a bitmap.

Many-to-many (n:n) migration. n:n migrations are implemented as an extension of n:1 migrations analogous to how 1:n migrations extend 1:1 migrations as described above. Thus, a hash table is used instead of a bitmap, but the migration bit for a group in the input table is only set once all of the group's dependent tuples in the output schema have been generated.

We call 1:1 and 1:n migrations "bitmap migrations", and n:1 and n:n migrations "hashmap migrations", and discuss each of these in more detail in the following subsections.

When the same input table is involved in separate migration statements, BullFrog maintains multiple data structures for it. For example, if a column is added to a table (1:1 migration) in addition to it generating a new table via a 1:n join (1:n migration), two bitmaps are allocated to manage the two migration operations on that table.

3.2 Migration transaction processing

The migration work precipitated by a client request is performed in a series of transactions that is separate from, and completed prior to, processing the client request transaction. Dividing work into multiple transactions simplifies abort handling and avoids deadlock.

Algorithm 1 shows BullFrog's per-worker logic for handling a client transaction during schema migration. Two worker-local lists, SKIP and WIP, start off empty (line 2). After starting a transaction, it iterates through the records in the old schema deemed to be potentially relevant via the predicate extraction process described in Section 2.1 (lines 3-4). For each relevant record, T is migrated if Algorithm 2 (bitmap migrations, Section 3.3) or Algorithm 3 (hashmap migrations, Section 3.4) says it should be (lines 5-6).

Algorithms 2 and 3 add to WIP tuples or groups for which they returned *true*, and add to SKIP those for which they returned *false* due to an existing migration effort by a different worker. After the migration transaction completes, the status of all tuples in WIP are updated to indicate that they have been migrated (lines 8-9) using the data structures to be discussed shortly. Finally, the loop body ends and the SKIP list is checked (line 10). If it is non-empty, the **do** loop repeats (in a fresh transaction) to recheck the status of these skipped tuples and migrate them in the rare case that the other worker that was migrating them aborted (see Section 3.5).

3.3 Bitmap migrations

For 1:1 and 1:n migrations, lock and migration status are tracked using two bits per migration granule, such as a tuple or page. (This section assumes tuple granularity for simplicity.)

- A **migrate bit** that is initialized to 0 and is set to 1 when that tuple has been migrated.
- A **lock bit** (or "in progress" bit) that is initialized to 0 and is set to 1 when a worker begins the process of migrating this tuple. Setting the bit to 1 prevents other migration workers from concurrently trying to migrate the same tuple.

These two bits are stored in adjacent positions in the bitmap so both can be accessed in a single read of a memory word. The top of Figure 1 shows an example bitmap, with [*lock-bit migrate-bit*] pairs associated with a set of 8 tuples. A pair of [0 0] in the bitmap indicates that the tuple has not yet started the migration process, [1 0] means that the migration is "in-progress", and [0 1] means the migration has completed. A state of [1 1] should never occur.

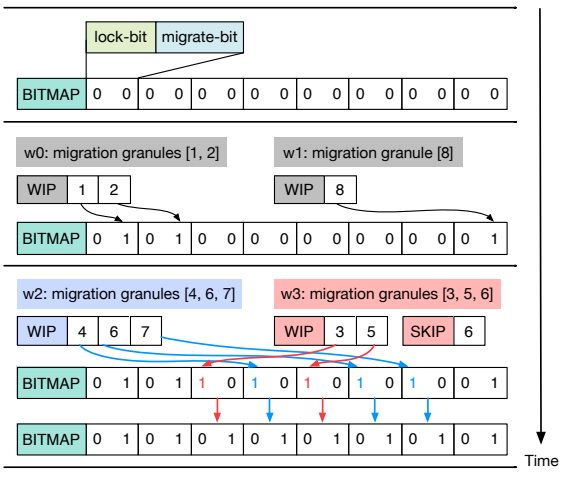


Figure 1: Schema migration during transaction processing.

Algorithm 2: Check bitmap whether to migrate a given tuple.

Input : shared bitmap `bitmap`, tuple index in bitmap `bkey`, local in-progress lists `WIP` and `SKIP`
Output: true, if the worker is permitted to migrate the tuple

```
1  if migrate bit of bkey in bitmap is not set then
2      if lock bit of bkey in bitmap is set then
3          append bkey to list SKIP
4          return false
5      acquire an exclusive latch on the bitmap
6      if migrate bit of bkey in bitmap is not set then
7          if lock bit of bkey in bitmap is not set then
8              set the lock bit of bkey in bitmap
9              release the latch on bitmap
10             append bkey to list WIP
11             return true
12         else
13             release the latch on bitmap
14             append bkey to list SKIP
15             return false
16     else release the latch on bitmap
17 return false
```

The bitmap is protected from concurrent workers by a read-write latch that allows concurrent reads but requires exclusive access for writes. We partition the bitmap into separate chunks protected by different latches to reduce cross-worker latch contention.

To check whether to migrate a particular tuple, the worker runs the pseudocode shown in Algorithm 2. Line 1 checks the migration bit of the tuple. If it is 1, it has already been migrated, so it returns false (line 17). Line 2 checks the lock bit. If it is 0, the code in lines 5-16 is run, that sets the lock bit to 1 and appends the tuple identifier into the in-progress list WIP of that worker. All of this is done after getting an exclusive latch on the bitmap partition and then rechecking the migration and lock bits to confirm they were not changed before the worker acquired the exclusive latch. If the lock bit is 1, another worker has already started the process of migrating this tuple. In that case, the code in lines 3-4 and 13-15 adds that tuple to the"skipped tuple" list SKIP for that worker.

Algorithm 1 migrates the tuple and at line 9 sets its status: [0 1].

The example in Figure 1 depicts four queries issued over the new schema, each of which spins up a worker to migrate relevant data (query q1 spins up worker w1, query q2 spins up worker w2, etc.). w0 executes simultaneously with w1, but they do not attempt to migrate the same data, and so can proceed independently. After this, w2 and w3 run concurrently and both attempt to read data located inside the 6th tuple. Although reads do not conflict with respect to data access, they do conflict with respect to migration workers, since only one worker is allowed to migrate this tuple. w2 acquires the lock bit on tuple 6 earlier than w3, so w2 appends tuple 6 to its WIP, while w3 observes that tuple 6 is locked and so appends it to its SKIP. When w3 finishes migrating tuples 3 and 5, it checks tuple 6 again to see if it was migrated. If it was, it allows q3 to run on the new schema. Otherwise, it blocks the query until tuple 6 is migrated or the lock is released.

Algorithm 3: Tuple eligibility checking for hash migrations.

Input : shared hash table `htable` and a tuple T, local in-progress lists `WIP` and `SKIP`
Output : true, if the txn is permitted to migrate T

```
1  Generate group key hkey from T
2  if hkey exists in list WIP then return true
3  if hkey exists in list SKIP then return false
4  if hkey exists in htable then
5      if hkey state in htable is in-progress then
6          append hkey into list SKIP and return false
7      if hkey state in htable is abort then
8          update the pair (hkey, in-progress) in htable
9          append hkey into list WIP and return true
10     return false
11 if htable.insert (hkey, in-progress) already exists then
12     GOTO LINE 7
13 else append hkey into list WIP and return true
```

3.4 Hashmap migrations

Both n:1 and n:n migrations require accessing multiple tuples from an input table in order to produce an output tuple. This means that tuple-level granularity tracking of migration status is inappropriate—either an entire group of tuples that combine to form an output tuple should be considered migrated, or none of them. BullFrog therefore tracks lock/migrate status at the group level for these migrations. Since mapping arbitrary group identifiers to unique offsets in a dense bitmap would be complex without advanced knowledge of the complete set of group identifiers, a hash table is used to track statuses, rather than a bitmap.

Given a tuple in an input table that is potentially relevant to a query result, Algorithm 3, line 1, determines the group identifier to use as a key into the hash table. For example, for a `GROUP BY` migration statement, the group identifier is constructed from the value of the attribute(s) that appear in the `GROUP BY` clause. With this key, the worker executes the remaining code in Algorithm 3.

Line 2: If the key exists in the list `WIP`, this means that this same worker has already decided to migrate a different tuple from the same group. Since the entire group must be migrated together, the worker must migrate the current tuple as well.

Line 3: If the key is found in the list `SKIP`, a different worker was already found to be migrating the group associated with this tuple and so it will be skipped by the current worker and revisited a later point to check whether the other worker successfully completed the migration of this group, as we described in Section 3.2.

Lines 4-10: If the key is found in the global hash table (but not the local lists), its current lock/migration status is checked. If it is locked but not yet migrated, this implies that the migration is `in-progress` by another worker, and the key is appended to the local list `SKIP`. If it is neither locked nor migrated, this implies that a different worker started the process of migrating it, but aborted. The worker thus (exclusively) updates the lock status to acquire the lock and appends the key to the local list `WIP`.

Lines 11-13: If the key is *not* found in the hash table, this implies that the data is neither locked nor migrated. The worker attempts to acquire a latch on the hash table[4] and insert the key with a value of "in progress" (locked but not migrated) into the hash table. If after acquiring the latch it finds that the key had already been inserted by another worker in between the initial check and the point where the latch was acquired, it releases the latch and runs the same code that would have been run if it had initially found the key in the hash table (line 12). Otherwise, the key is inserted into the local list `WIP` and the migration of that tuple can proceed (line 13).

Algorithm 1 performs the migration of the group, and updates its key in the hash table to a status of `migrated` at line 9.

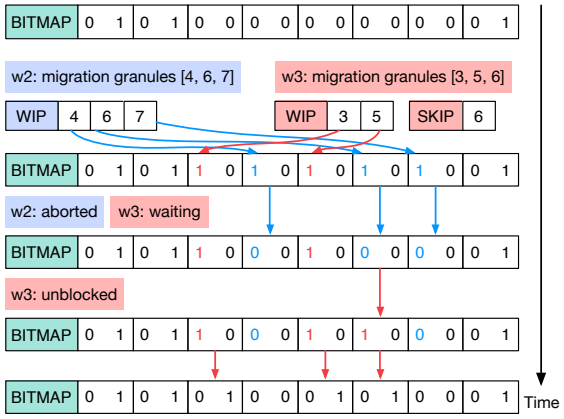


Figure 2: Transaction abort handling.

3.5 Migration aborts

When a migration transaction aborts, after the standard database system code is run to handle the abort, BullFrog[5] must inject additional code that traverses the aborted worker's `WIP` list, and for each element, the corresponding key is accessed in the bitmap or hash table and set to [0 0] in the bitmap or `abort` in the hash table.

Figure 2 depicts an example of abort handling for bitmap migrations. Workers w2 and w3 both access the 6th tuple in addition to other tuples. w2 accesses it first and grabs that lock and puts it in its `WIP`, while w3 sees that it is locked and puts it in its `SKIP`. After migrating tuples 4, 6, and 7, but before it commits, w2 gets aborted. This causes the underlying system to undo the insertion of the new tuples in the output tables that were caused by the migration of tuples 4, 6, and 7. At the end of the abort logic, BullFrog iterates through w2's `WIP` and resets the lock and migration statuses of those keys back to [0 0]. When tuple 6's status is reset to [0 0], w3 (which had been looping, waiting for the migration of tuple 6 to complete or abort) can migrate that tuple itself.

BullFrog's status tracking data structures are stored in volatile memory. Upon a crash, they must be reinitialized. While the REDO log is scanned during recovery, for each tuple (or group) that is found in a committed migration transaction, the corresponding status is set to [0 1] in the bitmap or `migrated` in the hashmap.[5]

[4] Similar to what we described in Section 3.3 for the bitmap, the hash table is partitioned and each partition is protected by a separate latch in order to reduce cross-worker contention that would arise if there were a global latch for the entire hash table. Deadlock does not occur since two latches are never acquired simultaneously.
[5] We have yet to implement this feature in the BullFrog codebase.

3.6 Joins

As we described in Section 3.1, joins can either be 1:1 migrations or 1:n migrations depending on the type of join. For example, a foreign-key/primary-key join is a 1:n migration relative to the primary key input table (PKIT), while at the same time being a 1:1 migration relative to the foreign-key input table (FKIT). However, we said for 1:n migrations, a tuple cannot be considered migrated until the n tuples that it generates have been migrated. Consider a tuple to migrate from the FKIT with a foreign key of 4. Since this is a 1:1 migration relative to the FKIT, the tuple from the PKIT with a primary key of 4 is extracted and joined with this tuple to produce the migrated version of it. After the migrated version has been successfully inserted into the output table, the input tuple in the FKIT can be marked in the bitmap as being migrated. However, the tuple in the PKIT that it joined with (the one with primary key of 4) cannot be considered migrated, since it is a 1:n migration and there may be other tuples in the FKIT with a foreign key of 4. There are two options for what to do next in such a scenario:
(1) *Immediately migrate all other tuples in the FKIT with the same foreign key.* This allows BullFrog to mark the key in the primary key table as `migrated` at the completion of this migration. However, this turns the 1:1 migration on the FKIT side into a n:n migration.
(2) *Stop at this point without adding any additional tuples to this particular migration task.* This option provides BullFrog with more flexibility to migrate lazily, and maintains the simpler 1:1 migration semantics on the FKIT. However, maintaining migration status for the PKIT requires occasional coordination with the FKIT to learn when all tuples with a particular value have been migrated.

In general, the second option is preferable when the cardinality of the foreign key is small or when there is skew such that large chunks of the FKIT would be forced to be migrated at once. In practice, when BullFrog uses the second option in the context of an inner join, it does not attempt to maintain the migration status of the PKIT. Furthermore, it does not maintain the lock status on the PKIT, since the unit of migration is entirely determined by individual tuples in the FKIT, and there are no semantic issues that arise when two different tuples from the FKIT are being migrated concurrently and access the same tuple from the PKIT. If an entire PKIT tuple is to be migrated, then all tuples from the FKIT that it joins with must be locked. Thus, there are no lock status or migration status bitmaps associated with the PKIT. Correspondingly, when using the first option in the context of an inner join, the unit of migration is entirely determined by individual tuples in the PKIT, so BullFrog does not maintain lock or migration bitmaps on the FKIT.

For many-to-many joins, BullFrog provides the same two options for maintaining the lock and migration status discussed above. However, both input tables are considered a 1:n migration with respect to the other table so it may be impossible to avoid migrating large chunks of data within individual migration tasks if there is skew in the attribute(s) involved in the join condition. Therefore, BullFrog also provides a third option of tracking status based on the combination of tuples from the two tables involved in the join, which increases the granularity of the lazy migration. I.e., instead of x.tupleID → (lock_status, migrate_status) it is (x.tupleID, y.tupleID) → (lock_status, migrate_status). BullFrog uses the hashmap technique described in Section 3.4 to track migration status.

3.7 Discussion: Conflict detection

In some cases, instead of using BullFrog's lock data structures, it would be possible to leverage the underlying database system to prevent duplicate migrations via SQL clauses such as `ON CONFLICT DO NOTHING`. Unfortunately, this method of preventing duplicate migrations has limited applicability. First, the output tables must have a uniqueness constraint declared on an attribute. Many database systems, such as PostgreSQL, require a B-tree index on any attribute declared to be unique. This uniqueness constraint must have been declared on a deterministic attribute whose value is based directly on values of data in the input table(s). Therefore, a primary key generated by an auto-increment function would not be eligible. Even though this primary key is unique, the duplicate insertion during the migration process will not be detected – instead the record will be inserted twice, with the system generating different unique primary keys for each record.

When applicable, this technique prevents the additional accesses to the old schema on behalf of migration transactions that are blocked, waiting for the old schema to be released. However, it detects conflict at a later stage (at the point of insert into the new schema) and therefore may incur additional wasted work upon a conflict. BullFrog supports both methods for handling migrations and we experimentally compare them in Section 4.

4 EXPERIMENTAL EVALUATION

We implemented a complete prototype of BullFrog on top of PostgreSQL 11.0. Our implementation leverages PostgreSQL's existing view expansion and query rewriting/optimization functionalities – we did not have to modify any core PostgreSQL code. Our bitmap data structures (see Section 3.3) use PostgreSQL's existing TIDs for mapping tuples to bits in the bitmap.

The primary goal of our experimental evaluation is to understand the downtime implications of *single-step* migration algorithms in which the database switches from the old to new schema immediately; such single-step migrations historically have required extensive downtime. To this end, we experimentally evaluate the *lazy migration* algorithms of BullFrog and compare their performance under various configurations against *eager migration*. In eager migration, the system immediately physically moves all data stored under the old schema into tables in the new schema prior to becoming available to client requests over the new schema.

In addition, we also benchmark BullFrog against a multi-step migration implementation in which a schema change is registered with the system ahead of time, and the system copies data into the new schema in a background process. Reads are served from the old schema, while writes go to both schemas. Although BullFrog is targeted for single-step deployment scenarios where giving advanced notification of a schema change is impossible, impractical, or simply too burdensome (see Sections 1 and 5), there are also some performance differences between single-step and multi-step algorithms that these experiments can illuminate.

For the BullFrog algorithms, we compare solutions that perform duplicate migration detection at time of insert into the new schema via PostgreSQL's `ON CONFLICT` clause (see Section 3.7) with solutions that detect duplication prior to generation of the migrated record (see Sections 3.3 and 3.4).

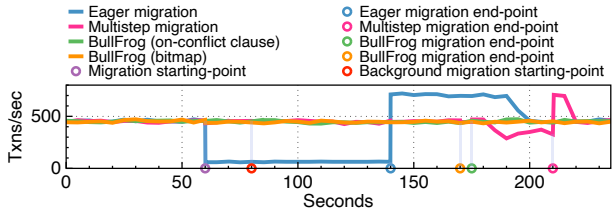
Workload. We developed a variation of TPC-C that includes schema migrations. TPC-C models the transactions involved in placing and delivering orders in a retail application; and querying stock levels of merchandise. The workload is defined by a mix of transactions according to the following percentages: `NewOrder` (45%), `Payment` (43%), `Delivery` (4%), `OrderStatus` (4%) and `StockLevel` (4%). `StockLevel` and `OrderStatus` are external read queries. The schema defined by the TPC-C benchmark consists of nine tables. Our experiments evolve the original schema in various ways, the specifics of which will be discussed in the following sections.

Experimental Platform. We use OLTP-Bench [21] to set up and run our experiments. OLTP-Bench has the ability to support tight control of transaction mixtures, request rates, and access distributions over time. We measure throughput as transactions per second and the end-to-end latency as the time from when the client issues a transaction request until the response is received. Maximum throughput measurements are taken by increasing the rate that clients submit requests until the latency of these requests starts to increase due to queuing delays. The measurements for all of our throughput experiments are averaged over 10 runs, but we found that the variance across runs in each of our experiments was negligible. Latency experiments are presented using cumulative distribution functions (CDFs), and each plot shows a distribution over at least 50,000 points. We run our experiments on an eight-core 2.50GHz Intel Core i7 using 16GB of memory. We dedicate all eight cores to workers within the transaction processing engine.

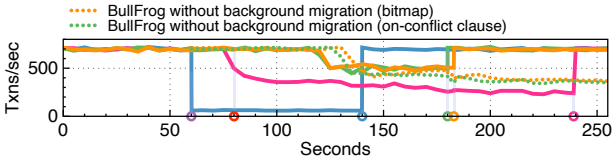
4.1 Table split migration

In our first experiment, the baseline TPC-C schema incurs a relatively simple migration: the customer table is split into two tables, where its original set of columns are divided across the two new tables, except for the primary key which appears in both new tables. Each new table contains the same number of tuples as the original customer table. One table includes the customer's personal financial information (balance, payment, credit, etc.) and the other has the customer's less private information (city, state, zip). Using the terms we described in Section 3.1, this is a 1:n migration with respect to the original customer table, since for every row in the customer table there are two rows generated (one row for each of the new tables). Therefore, BullFrog uses a bitmap data structure to track the migration. In this experiment, we use the TPC-C configuration with 50 warehouses, which therefore causes the benchmark to generate 1.5 million records in the customer table. Four out of the five TPC-C transaction types access the customer table—`NewOrder`, `Payment`, `Delivery` and `OrderStatus` and are straightforwardly modified to be compatible with the new customer tables.

Figure 3 shows how the throughput of transaction processing varies during the different phases of the schema migration. Figure 3(a) shows the common case where the system is not overloaded at the time of the migration, and the system can devote extra resources to the additional work involved in migrating to a new schema. Figure 3(b) shows the same experiment, except that the clients are already submitting requests at the maximum rate at which the system is able to keep up (without falling behind) before the migration, and thus the additional migration work necessarily forces



(a) Request rate: 450 transactions per second.



(b) Request rate: 700 transactions per second.

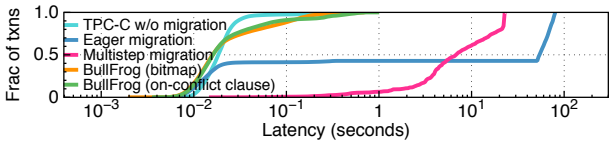
Figure 3: Throughput during table-split migration.

the system to fall behind. The migration begins for all implementations at the purple circle and ends for each system at the later corresponding circles marked in the figure.

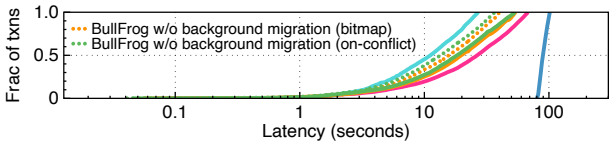
Eager migration takes approximately 80 seconds to complete. This time is independent of client request load because all requests that access the customer table during the migration are queued, and the performance of the migration itself is not affected by the size of this queue of waiting transactions. Throughput does not dip all the way to 0 since the `StockLevel` transaction does not access the customer table and can be processed even during an eager migration. When the client load is 450 transactions per second (TPS), there is enough system headroom for the eager migration system to catch up after the migration. This observed in Figure 3(a) by a temporary increase in throughput after the migration relative to the throughput before the migration began. When the client load is maxed out at 700 TPS (Figure 3(b)), the eager system can never catch up after the migration and cannot decrease the size of the request queue that built up during the migration.

For the lazy migration algorithms, the total time to complete the migration is longer since they process active client requests concurrently with performing the migration. Nonetheless, the background process discussed in Section 2.2 enables the migration to complete within the time window shown in the figure. Without the background process (the dotted lines in the figure), the TPC-C benchmark does not access enough distinct tuples to complete the migration within the experimental time window. Throughput steadily degrades as the tables in the new schema become larger and slower to access, while access costs to the fixed-size old schema remains constant despite the increasing percentage of transactions that find out that all relevant tuples have already been migrated.

Lazy migration throughput is unaffected by the migration when the client request rate is at 450 TPS. The additional per-transaction overhead of lazily migrating relevant records to the new schema tables is hidden by the spare capacity of the system. However, when the client load is maxed out at 700 TPS, the throughput is ultimately affected by the migration. At first, the migration overhead is visible in transaction latency, but throughput is not affected. Eventually, OLTP-Bench is forced to queue transactions before sending them to PostgreSQL, and the throughput drops along with the additional

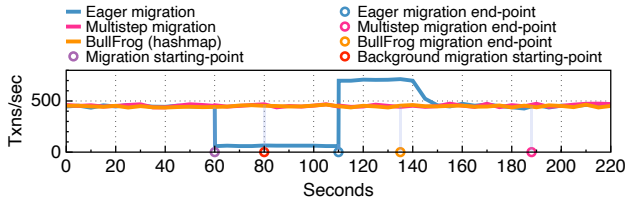


(a) Request rate: 450 transactions per second.

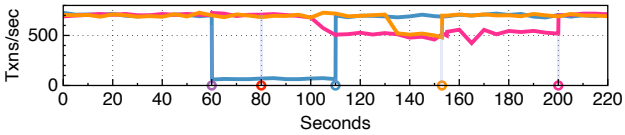


(b) Request rate: 700 transactions per second.

Figure 4: Latency during table split migration.



(a) Request rate: 450 transactions per second.



(b) Request rate: 700 transactions per second.

Figure 5: Throughput during aggregation migration.

queuing latency. The performance of the bitmap vs. on-conflict approaches are similar. In addition to the more steady throughput curves, the lazy migration also performs 13% more transactions overall than the eager migration approach during this experimental window. We attribute this additional efficiency to the improved cache efficiency of bringing in a tuple into cache just once to both migrate it and use it as part of an active client request.

For lazy migration, background migration threads do not begin until 20 seconds after migration initiates, since at first, the client requests themselves are sufficient to keep the migration progress moving along. Only later do the background threads start and search for data to migrate that has not yet been covered by a client request. In contrast, for multi-step migration, the entire migration process happens in the background. Therefore, the background threads start immediately which causes an earlier performance drop.

Surprisingly, in contrast to lazy migration in which the background threads help accelerate the completion of the migration, multi-step migration takes **longer** to complete than lazy migration despite the presence of background threads throughout the migration. The reason is that during the early stages of multi-step migration, most data exists only in the old schema, and updates are only performed in the old schema. However, as migration continues, a larger percentage of data has been migrated to the new schema. Any updates to migrated data must happen twice – in the new and old schema – since the old schema must be able to serve reads until the migration completes. Therefore, as the migration progresses, the multi-step migration needs to perform additional work relative to lazy migration (which never has to perform updates on the old schema). This is observed in the experiments by a steadily dropping throughput for multi-step migration, until the migration completes.

Figure 4 shows a CDF of client request latency for the same experiment, starting at the point the migration begins until the end of the experimental window from Figure 3. Latency results are plotted for only one transaction type (the most complex – `NewOrder`) in order to avoid variations due to the different complexities of different TPC-C transaction types. When the client request rate is 700 TPS, the eager migration algorithm is never able to catch up. Thus, the 80 second downtime required to perform the migration is experienced not only by the requests that were submitted during the downtime, but also by the requests that were submitted afterwards, since the size if the request queue never has a chance to decrease. In contrast, at 450 TPS, the eager system is able to catch up. Therefore, the CDF appears as a step – the left side of the graph shows the requests that were submitted after the system catches up, while the right side shows the latency of transactions before the system catches up. The multi-step and lazy schemes, are also never able to catch up when then are no spare resources in the system at 700 TPS. However, because of its superior throughput, BullFrog never gets as far behind as the other algorithms, and both the BullFrog and multi-steps algorithms fall behind at a more steady rate because of their lack of downtime. The poor latency for multi-step at 450 TPS is caused by the throughput dip from Figure 3 and resulting queuing delays. Overall, the latency of the lazy algorithms is up to an order of magnitude better than the eager and multi-step algorithms and is comparable to the latency of TPC-C without any migration.

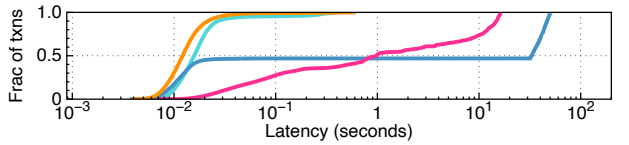
4.2 Aggregate Migration

The `Delivery` transaction collects a number of the oldest undelivered orders and marks them as having been delivered. As part of this process, it performs an implicit aggregate operation as follows:

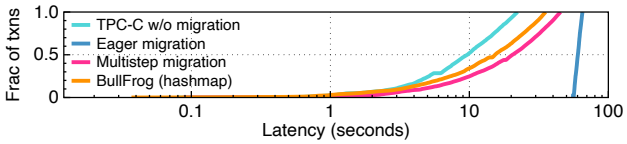
```
SELECT SUM(OL_AMOUNT) AS OL_TOTAL FROM ORDER_LINE
    WHERE OL_O_ID = ? AND OL_D_ID = ? AND OL_W_ID = ?;
```

In our next experiment, we model a schema evolution in which this aggregation is maintained as a separate table. This evolution can be thought of as a materialized view that is maintained by the application instead of the database system. The migration request runs the initial aggregation of the 15 million tuples in the `ORDER_LINE` table, and all future transactions update both the original and aggregated version of this table.

Figures 5 and 6 show the throughput and latency of this experiment using the same methodology as Figures 3 and 4 respectively. Using the terminology from Section 3.1, this is a n:1 migration with respect to the `ORDER_LINE` table (in contrast to the previous experiment which was a 1:n migration). Therefore, BullFrog uses a hashmap data structure instead of a bitmap to track the migration. Nonetheless, the results of this experiment are similar to the table-split experiment where throughput and latency are not affected by the migration at 450 TPS, but all systems fall behind at 700 TPS. However, the amount of data that must be written as part of the aggregation migration is smaller (since the output table is small), so the migration is cheaper and the window of throughput reductions for all approaches is smaller and the systems fall less behind.

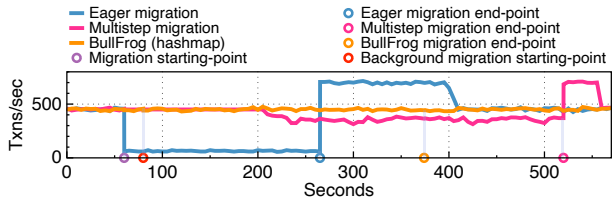


(a) Request rate: 450 transactions per second.

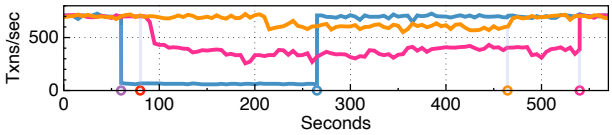


(b) Request rate: 700 transactions per second.

Figure 6: Latency during aggregation migration.



(a) Request rate: 450 transactions per second.



(b) Request rate: 700 transactions per second.

Figure 7: Throughput during join migration.

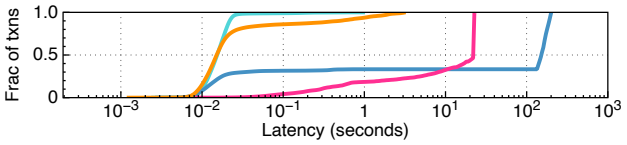
4.3 Join Migration

TPC-C's `StockLevel` transaction is a read-only transaction that scans a warehouse's inventory for items which are (or close to being) out of stock. As part of this process, a join occurs:

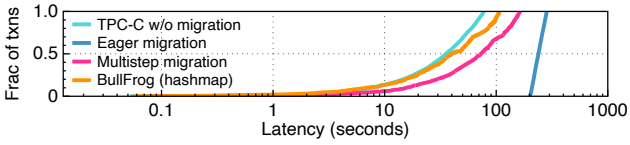
```
SELECT COUNT(DISTINCT (S_I_ID)) AS STOCK_COUNT
    FROM ORDER_LINE, STOCK WHERE S_I_ID = OL_I_ID ...;
```

We model a situation where the application developers prioritize the performance of `StockLevel` queries by denormalizing the schema so that the order line and stock tables are already joined. The new schema includes this new table – named `orderline_stock` – instead of the original order line and stock tables. All transactions that accessed the old tables are replaced by new transactions against the `orderline_stock` table that consists of close to 8 million tuples.

Figure 7 shows the throughput results of this experiment and Figure 8 shows the latency results. This join is the most resource intensive of all the migrations we have experimented with, and thus the throughput dip of all systems, including multi-step migration, is more extended (except BullFrog at 450 TPS which still has no throughput dip). The eager approach experienced over 200 seconds of downtime. The join involved in the migration is a many-to-many join, and BullFrog uses the hashmap-based n:n migration approach discussed in Section 3.6. When BullFrog attempts to perform the migration during a period of maximum load (700 TPS), latency steadily increases to 10 seconds per transaction, until PostgreSQL



(a) Request rate: 450 transactions per second.



(b) Request rate: 700 transactions per second.

Figure 8: Latency during join migration.

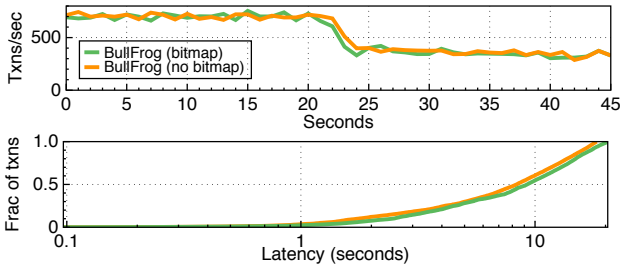


Figure 9: Data structure maintenance cost.

reaches its maximum number of concurrent transactions. At this point, throughput dips by approximately 100 TPS as OLTP-Bench queues transactions. After the migration completes, the throughput returns to its original level since the new pre-joined table is designed to accelerate the `StockLevel` transaction which appears relatively infrequently in TPC-C (4% of transactions). Furthermore, the `orderline_stock` table retains all secondary indexes of the two tables that generated it. However, latency never returns to its original level since the system is running at maximum load and can never catch up. The same is true of the eager system, but since it got further behind, the steady latency after the migration is an order of magnitude higher than BullFrog. Thus the lazy approaches are superior to the eager and multi-step approaches, both in terms of the size of the throughput dip and also the increase in latency.

4.4 Tracking Overhead

We now investigate some sources of overhead in BullFrog. All experiments in this section use the table split migration.

4.4.1 Data Structure Maintenance. We first measure the overhead data structure maintenance in BullFrog by comparing BullFrog performance with a version where no data structures are necessary. Instead, the application is modified such that the `NewOrder` transactions cumulatively access each tuple in the old schema exactly once, rendering migration status tracking unnecessary. Figure 9 shows the throughput and latency improvements of removing the tracking data structures is small since they do not introduce significant overhead.

4.4.2 Lock and Latch Contention. In this experiment, we create a variable number of *hot records* over which transactions exclusively access. Decreasing the size of this hot set increases the contention

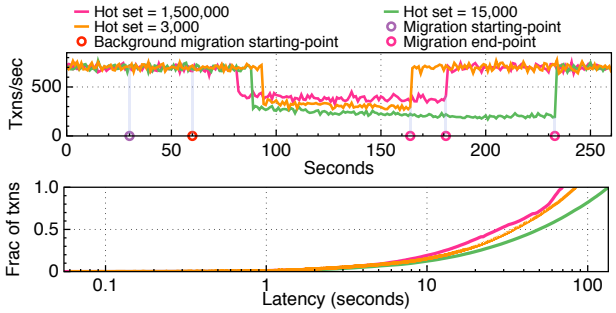


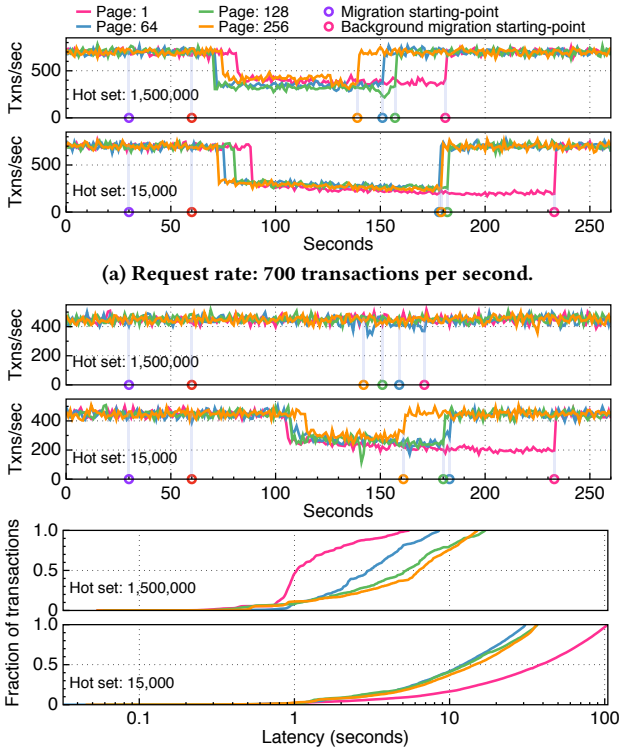
Figure 10: Skewed data access.

in the workload, and causes two potential problems for BullFrog. First, it increases the probability of duplicate, simultaneous attempts to migrate a tuple, which causes one of them to block until the first one completes the migration. Second, it increases latch contention for the hot partitions in BullFrog's data structures. Figure 10 shows that decreasing the hot set from 1,500,000 to 15,000 records indeed causes a longer drop in throughput during the migration. We verified that this was due to lock contention (and not latch contention) by rerunning the same experiment without making transactions wait upon reaching a locked record. We found that the performance drop was due to transactions repeating the loop through the set of records to migrate, waiting for the lock to be released (line 10 from Algorithm 1). This increases the latency of the transaction and reduces the number of transactions that can processed concurrently, thereby extending the migration time. However, for very small hot sets, the opposite phenomena is observed. The hot set gets quickly migrated, and the rest of the migration is performed by the background threads which can proceed efficiently and independently with minor impact on throughput.

4.4.3 Migration Granularity. We next vary the granularity of migration. Instead of tracking migration status at the tuple level, it is done at the page level, and we vary the size of pages and contention in the workload. Figure 11 shows that migrating data in larger chunks increases the latency of each operation, but allows the migration to complete more quickly. At 450 TPS, single-tuple granularity is best at low contention, since no matter the granularity, BullFrog can keep up with the request rate, so the latency advantage of fine granularity is preferable. However, under higher contention, coarse granularity migration is preferred, since the latency benefit of migrating with tuple granularity is negated by the additional queuing delays caused by the extended migration period. This is also the case when running at 700 TPS.

4.5 Integrity Constraints

We next evaluate the overhead of constraint preservation during a migration. The TPC-C benchmark includes foreign key constraints from the `Customer` table to `Order` and `District`. In Figure 12(a) we remove one (green line) or both (pink line) constraints to observe the improvement in performance from avoiding the overhead of migrating additional data in order to check the constraints. As explained in Sections 2.1, 2.3, and 2.4, the presence of integrity constraints in the new schema limits the laziness in which BullFrog can work, since it must migrate not only the data being accessed by



(a) Request rate: 700 transactions per second.

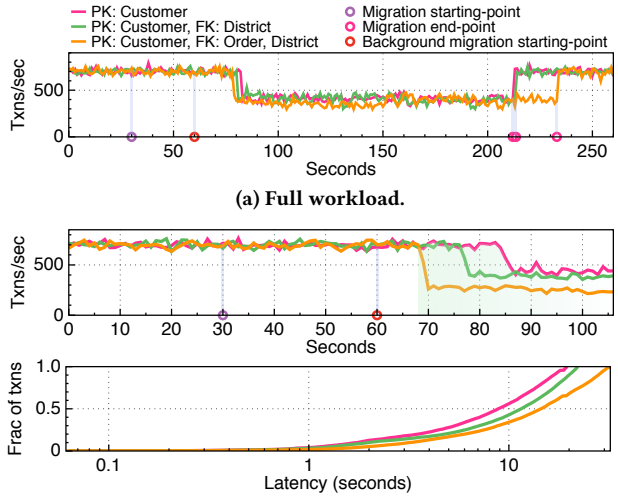
(b) Request rate: 450 transactions per second.

Figure 11: Varying access skew and migration granularity. End points are marked by the corresponding circles.

the client request, but also all data necessary to check the integrity constraints in the new schema. Since not all transactions in TPC-C access the customer table, the difference in performance was hard to observe. Therefore, we repeated the same experiment, but removed the transactions that do not access the customer table from the workload. These results are presented in Figure 12(b), where the overhead of constraint preservation manifests primarily as an earlier drop in throughput. This is because the additional data that is migrated per transaction limits the number of transactions that can be processed concurrently which accelerates the point at which the DB pushes back on OLTP-Bench to reduce the input workload.

5 RELATED WORK

Ronstrom [32] describes online schema evolution using triggers in a telecom database which is not allowed to be down for more than a minute in a year. The paper proposes two different ways to update the schema: soft schema change and hard schema change. For soft schema changes, old transactions are executed using the old schema and new transactions are processed using new schema. Old and new transactions can execute concurrently. For hard schema changes, transactions in the old schema are executed until all of them have finished executing. Thereafter the system switches over to using the new schema. However, even the hard switch uses a multi-phase process in which triggers are used to prepare the new schema in advance of the switch. In contrast, BullFrog supports single-step schema evolution.



(a) Full workload.

(b) Partial workload.

Figure 12: `FOREIGN KEY` constraints on table split migration.

The work on non-blocking schema change in F1 by Google works similarly to the "soft schema change" mechanism described by Ronstrom [31]. Schema changes are done asynchronously across servers. Since F1 is a distributed system with no synchronization across the servers, different servers may transition to the new schema at different times and multiple schema versions may be in use simultaneously. To simplify reasoning about correctness of the implementation, the authors restrict the servers in an F1 instance from using more than two distinct schema versions. Tools such as LessQL [10] facilitate automatic rewriting of queries to use evolved versions of a schema. However, soft schema change solutions restrict the scope of the schema evolution to ensure capability across all active schema versions. In contrast, BullFrog uses a more general approach that does not restrict the scope of the migration operations.

A host of schema migration tools generalize the state-of-the-art multi-step schema migration process, including Percona online schema change [5], Facebook online schema change [1], OAK online alter table [2] and LHM [3]. In the first step, the new schema is registered without yet becoming actively used. Writes performed to a source table (from the old schema) are propagated into a shadow table (for the new schema) that is gradually synchronized in the background using triggers. After all the data has been migrated, the second step involves switching over to the new schema by locking the source table briefly and renaming the shadow table (if necessary) to bring it online. The work on QuantumDB [20] along with the dissertation by Zhu [40] use a similar approach which use a combination of materialized views and triggers for maintaining consistency with the original tables while they are updated. In addition to the general disadvantages of multi-step migrations that we discussed in the introduction, all of these tools use update/insert triggers for applying the changes from the old table to the new table. Triggers are known to increase lock contention and at times render the table or the entire database inaccessible due to contention [6].

Løland and Hvasshovd [26] avoid the use of triggers by using log propagation to perform non-blocking schema transformation. Similarly, Github's online schema change tool `gh-ost` slowly and incrementally copies existing data from the source table to the shadow table while using the binary log stream in MySQL to capture ongoing changes on the source table, and replaying them to the shadow table asynchronously [4]. When the write load gets higher, `gh-ost` can't keep up with binary log, and may not finish at all [7]. Since these techniques read from the log files, there is a delay in between the time the changes are committed in the original table and the time they are applied to the shadow table. Similar to the other multi-step migration techniques we discussed above, these techniques allow queries to execute over the new schema only after the shadow table is caught up to the source.

Schema migration shares some complexities with database migration in which a database is moved from a source node to a destination node, and the copy on the source node is either kept or deleted after the migration [8, 12, 19, 27]. There also exists lazy implementations of database migration, in which data is pulled to the destination node as it is needed [22, 35], with background processes that ensure all data is eventually migrated, similar in theme to BullFrog. However, these lazy approaches do not make significant changes to the schema during migration. At most, simple 1:1 schema migrations are allowed such as type changes of an attribute. In contrast, BullFrog implements lazy schema migration that supports an arbitrary number of complex schema changes, including 1:n, n:1, and n:n migrations.

Our goal of single-step schema evolution is driven by the software maintenance community. For example, the work on KVolve starts with the same single-step migration requirement [33] and uses a lazy migration approach in the context of migrating an application on top of a NoSQL database (Redis). NoSQL databases are widely used by continuous deployment practitioners since they typically do not enforce schema constraints. Our work on BullFrog proves that lazy migration can be used for traditional relational database systems that enforce schema constraints and require physical data reorganization during a schema migration.

BullFrog's lazy approach to schema migration can be thought of as combining previous work on lazy transaction processing in database systems [24] with transaction decomposition [23, 37, 39] such that a large migration transaction is decomposed into separate smaller transactions that are processed lazily.

6 CONCLUSION

As applications and database systems increasingly evolve in lockstep, the database system must support single-step migration where the database must instantaneously switch over to a new schema with no downtime. BullFrog succeeds in using a lazy migration approach so that the new schema can be instantaneously ready for access even when the physical data has not yet been migrated to the new schema. Experiments show that BullFrog's lazy approach only causes a slight reduction in throughput and increase in latency during the migration, in contrast to eager approaches that cannot process any transactions during the migration.

7 ACKNOWLEDGEMENTS

This work was sponsored by the NSF under grants IIS-1718581 and IIS-1910613. We thank the anonymous reviewers and Karla Saur for their extremely helpful feedback and ideas.

REFERENCES

[1] 2010. Facebook Online Schema Change. https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932/.

[2] 2010. OAK Online Alter Table. https://shlomi-noach.github.io/openarkkit/oak-online-alter-table.html.

[3] 2012. Large Hadron Migrator. https://github.com/soundcloud/lhm.

[4] 2016. GitHub Online Schema Change. https://github.com/github/gh-ost.

[5] 2016. Percona Online Schema Change. https://www.percona.com/doc/percona-toolkit/2.2/pt-online-schema-change.html.

[6] 2016. Why Triggerless? https://github.com/github/gh-ost/blob/master/doc/why-triggerless.md.

[7] 2017. Gh-ost benchmark against pt-online-schema-change performance. https://www.percona.com/blog/2017/07/12/gh-ost-benchmark-against-pt-online-schema-change-performance/.

[8] 2020. Database migration: Concepts and principles (Part 2). https://cloud.google.com/solutions/database-migration-concepts-principles-part-2.

[9] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[10] Ariel Afonso, Altigran da Silva, Tayana Conte, Paulo Martins, João Cavalcanti, and Alessandro Garcia. 2020. LESSQL: Dealing with Database Schema Changes in Continuous Deployment. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 138–148.

[11] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342.

[12] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. 2012. "Cut me some slack" latency-aware live migration for databases. In *Proceedings of the 15th international conference on extending database technology (EDBT)*. 432–443.

[13] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A software architect's perspective*. Addison-Wesley Professional.

[14] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7. Portland, OR, 343477–343502.

[15] Eric A Brewer. 2001. Lessons from giant-scale services. *IEEE Internet Computing* 5, 4 (2001), 46–55.

[16] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. 2015. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology* 57 (2015), 21–31.

[17] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *The VLDB Journal* 22, 1 (2013), 73–98.

[18] Carlo A Curino, Hyun J Moon, MyungWon Ham, and Carlo Zaniolo. 2009. The PRISM workwench: Database schema evolution without tears. In *IEEE 25th International Conference on Data Engineering (ICDE)*. IEEE, 1523–1526.

[19] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.

[20] Michael de Jong, Arie van Deursen, and Anthony Cleve. 2017. Zero-downtime SQL database schema evolution for continuous deployment. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 143–152.

[21] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[22] Aaron J Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 301–312.

[23] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[24] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. 2014. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 15–26.

[25] Christopher M Hayden, Karla Saur, Edward K Smith, Michael Hicks, and Jeffrey S Foster. 2014. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 4 (2014), 1–38.

[26] Jørgen Løland and Svein-Olaf Hvasshovd. 2006. Online, non-blocking relational schema changes. In *International Conference on Extending Database Technology (EDBT)*. Springer, 405–422.

[27] Takeshi Mishima and Yasuhiro Fujiwara. 2015. Madeus: database live migration middleware under heavy workloads for cloud environment. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 315–329.

[28] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 573–585.

[29] Luís Pina, Luís Veiga, and Michael Hicks. 2014. Rubah: DSU for Java on a stock JVM. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, Vol. 49. 103–119.

[30] Dong Qiu, Bixin Li, and Zhendong Su. 2013. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 125–135.

[31] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1045–1056.

[32] Mikael Ronstrom. 2000. On-line schema update for a telecom database. In *Proceedings of 16th International Conference on Data Engineering (ICDE)*. IEEE, 329–338.

[33] Karla Saur, Tudor Dumitraş, and Michael Hicks. 2016. Evolving nosql databases without downtime. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 166–176.

[34] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 21–30.

[35] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. 2013. Prorea: live database migration for multi-tenant rdbms with snapshot isolation. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*. 53–64.

[36] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. 2017. Beyond continuous delivery: an empirical investigation of continuous deployment challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 111–120.

[37] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 325–363.

[38] Michael Stonebraker, Dong Deng, and Michael L Brodie. 2016. Database decay and how to avoid it. In *IEEE International Conference on Big Data (Big Data)*. IEEE, 7–16.

[39] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 276–291.

[40] Yu Zhu. 2017. *Towards Automated Online Schema Evolution*. Ph.D. Dissertation. UC Berkeley.