Appendix II. Formal proof

We apologize for the length of this appendix, and request the PC's indulgence in reading what follows.

We paraphrase the inductive invariants (I1, I2 and I3, Section 3.4) into Lemma 1, 5 and 6, and the correctness condition (Section 3.4) into Theorem 1. We show that Theorem 1 can be proved by Lemma Lemma 1, 5 and 6, and finally we prove the linearizability (Theorem 2) based on Theorem 1.

In the following proof, we do not discuss concurrent compaction and structure update as they are performed on a group exclusively in the implementation. We assume `get_group` always returns the correct group such that its pivot key is less than or equal to given key and with its next group's pivot key is greater than given key. Since remove can be treated as a special case of put (set removed flag instead of change value, Section 3.3), we ignore it in the discussion. Since model split/merge and group merge can be seen as a simplified version of group split, we also omit them in the discussion.

Definition 1. (commit/linearization points) We say a `put` commits if, depending on the implementation states, `put` executes line 25, 28, 32 or 34 of Algorithm 2 (the linearization points). Specifically, when executing line 28, 32 or 34, the linearization point is in between the invocation. For the vanilla (temporary) delta index implementation using a big lock, it is the point when the lock is aquired. `put` commits at.

- line 25 when it updates a record in sorted array;
- line 28 when it inserts/updates a record in delta index;
- line 32 when it updates a record in frozen delta index; and.
- line 34 when it inserts/updates a record in temporary delta index.

We say a `get` commits if, depending on the implementation states, `get` executes line 5, 10 or 12 of Algorithm 2 (the linearization points). Specifically, when executing line 10, 12, the linearization point is in between the invocation. For the vanilla (temporary) delta index implementation using a big lock, it is the point when the lock is aquired. `get` commits at.

- line 5 when it reads a record in sorted array;
- line 10 when it reads a record in delta index;
- line 10 when it fails to read a record in delta index and temporary delta index is not initialized;
- line 12 when it reads a record in temporary delta index; and.
- line 12 when it fails to read a record in temporary delta index and temporary delta index is initialized.

For two operations a and b, we use a ≪ b denote that a commits before b commits.

Definition 2. (conflicting operations) If operation a and b are invoked with the same key and one of them is `put`, then we say a and b are conflicting operations.

Definition 3. (phase transition) We say a `compaction` or `group_split` is performing phase transition if line 9-10, Algorithm 3 or line 9-10, 16-17, Algorithm 4 is being executed by a background thread.

It is obvious that, for given key, at most two groups are visible (returned by `get_group`) to threads during `compaction` or `group_split` performing phase transition. We term the group before phase transition as the *old group* and the created one as *new group*.

Definition 4. (record copy) A record copy in XIndex is a pair of key and value that.

- is stored in either `data_array`, `buf` or `tmp_buf`; and.
- the `data_array/buf/tmp_buf` can be accessed via the global root pointer and group pointers observed by an operation (i.e., returned by `get_group`).

If a record copy contains key k, then we say it is a record copy of k.

Lemma 1. (I1) For any possible concurrent execution of XIndex, for any key k, if there is `put` committed on k, then there is exactly one record copy of k in XIndex.

Proof of Lemma 1.

- 1 If there is `put` committed on k, then there is at least one record copy of k in XIndex.
  - 1.1 If there is no record copy of k in XIndex before put commits, then a new record copy of k will be created.
    - 1.1.1 If there is no record copy of k in XIndex before put commits, then put commits at line 28 or line 34, Algorithm 2.
      - 1.1.1.1 data_array is un-insertable after initialization.

        By the implementation of `get_position` that returns empty when there is no record copy of k in `data_array`.
      - 1.1.1.2 buf cannot be in-place updated.

        By the implementation of `try_update_in_buffer` that fails when there is no record copy of k in the buffer.
      - 1.1.1.3 Q.E.D.
    - 1.1.2 If put commits at line 28 or line 34, Algorithm 2, then a new record copy of k will be created.

      By the implementation of `insert_buffer` that creates a new record copy of k when there is no record copy of k in the buffer.
    - 1.1.3 Q.E.D.
  - 1.2 Other operation does not remove record copies.

    By OBVIOUS.
  - 1.3 Q.E.D.
- 2 If there is `put` committed on k, then there is at most one record copy of k in XIndex.
  - 2.1 It suffices to prove that for any key k, 1) two record copies of k can not exist within one group; and 2) two record copies of k can not exist in two different groups.

    By OBVIOUS.
  - 2.2 Two record copies of k can not exist within one group.
    - 2.2.1 Two record copies of k can not exist in the different locations of a group.
      - 2.2.1.1 **CASE 1:** Two record copies of k can't exist in *data_array* and *buf* respectively.
        - 2.2.1.1.1 Record copy in buf is created by `insert_buffer`.

          By line 28 and 34, Algorithm 2; line 8, Algorithm 3; line 6, 14, Algorithm 4.
        - 2.2.1.1.2 **CASE 1**: Record copy created by inserting to buf.
          - 2.2.1.1.2.1 In case of 2.2.1.1.2, when put creates a record copy in buf, there are two record copy in data_array and buf respectively when put commits.

- 2.2.1.1.2.1.1 Record copy in data_array is created before record copy in buf.
  - 2.2.1.1.2.1.1.1 *data_array* can't be inserted after initialization.

    By the implementation of data_array that provides no insert interface.
  - 2.2.1.1.2.1.1.2 Only after initializing a group, put can insert new records to buf.

    By the implementation that we first initialize buf of new group at line 8, Algorithm 3 and line 6, 14, Algorithm 4 before making group visible to threads at line 9-10, Algorithm 3 and line 9-10, 16-17, Agorithm 4.
  - 2.2.1.1.2.1.1.3 Q.E.D.
- 2.2.1.1.2.1.2 Q.E.D.

- 2.2.1.1.2.2 If put creates a record copy in buf, there is no record copy in data_array when put commits.

  By Lemma 2.
- 2.2.1.1.2.3 2.2.1.1.2.1 and 2.2.1.1.2.2 contradicts.

  By OBVIOUS.
- 2.2.1.1.2.4 Q.E.D.

- 2.2.1.1.3 **CASE 2**: Record copy created by inserting to tmp_buf of old note and later become the buf of new group.
  - 2.2.1.1.3.1 In case of 2.2.1.1.3, when put creates a record copy in tmp_buf, there are two record copy in data_array/buf and tmp_buf of the old group respectively when put commits.
    - 2.2.1.1.3.1.1 Record copy in data_array/buf of old group is created before record copy in buf.
      - 2.2.1.1.3.1.1.1 The record copy in new group's data_array is merged from old group's data_array or buf.

        By line 7, Algorithm 3; line 13, Algorithm 4.
      - 2.2.1.1.3.1.1.2 The old group's data_array/buf is not insertable after old group's tmp_buf is initialized.

        By line 3-5, Algorithm 3; line 2-3, 7, Algorithm 4.
      - 2.2.1.1.3.1.1.3 put inserts record copy in tmp_buf after tmp_buf is initialized.

        By OBVIOUS.
      - 2.2.1.1.3.1.1.4 Q.E.D.
    - 2.2.1.1.3.1.2 Q.E.D.
  - 2.2.1.1.3.2 If put creates a record copy in tmp_buf, there is no record copy in data_array or buf when put commits.

    By Lemma 3.
  - 2.2.1.1.3.3 2.2.1.1.3.1 and 2.2.1.1.3.2 contradicts.

    By OBVIOUS.
  - 2.2.1.1.3.4 Q.E.D.
- 2.2.1.1.4 Q.E.D.

- 2.2.1.2 **CASE 2**: Two record copies of k can't exist in *data_array* and *tmp_buf* respectively.
  - 2.2.1.2.1 Record copy in *tmp_buf* is created by `insert_buffer`.

    By line 28 and 34, Algorithm 2; line 8, Algorithm 3; line 6, 14, Algorithm 4.
  - 2.2.1.2.2 In case of 2.2.1.2, when put creates a record copy in tmp_buf, there are two record copy in data_array and tmp_buf respectively when put commits.
    - 2.2.1.2.2.1 Record copy in data_array is created before record copy in tmp_buf.
      - 2.2.1.2.2.1.1 *data_array* can't be inserted after initialization.

        By the implementation of data_array that provides no insert interface.
      - 2.2.1.2.2.1.2 Only after initializing a group, put can insert new records to tmp_buf.

        By the implementation that we always initialize tmp_buf to a group that is already initialized (line 8, Algorithm 3 and line 14, Algorithm 4).
      - 2.2.1.2.2.1.3 Q.E.D.
    - 2.2.1.2.2.2 Q.E.D.
  - 2.2.1.2.3 If put creates a record copy in tmp_buf, there is no record copy in data_array when put commits.

    By Lemma 3.
  - 2.2.1.2.4 2.2.1.2.2 and 2.2.1.2.3 contradicts.

    By OBVIOUS.
  - 2.2.1.2.4 Q.E.D.
- 2.2.1.3 **CASE 3:** Two record copies of k can't exist in *tmp_buf* and *buf* respectively.
  - 2.2.1.3.1 Record copy in buf and *tmp_buf* are created by `insert_buffer`.

    By line 28 and 34, Algorithm 2; line 8, Algorithm 3; line 6, 14, Algorithm 4.
  - 2.2.1.3.2 In case of 2.2.1.3, when put creates a record copy in tmp_buf, there are two record copy in buf and tmp_buf respectively when put commits.
    - 2.2.1.3.2.1 If there is only one record copy, then the record copy in buf can't be created after the put that created record copy in tmp_buf commits.
      - 2.2.1.3.2.1.1 buf is not insertable after tmp_buf is initialized.

        By line 3-5, Algorithm 3; line 2-3, 7, Algorithm 4.
      - 2.2.1.3.2.1.2 The put that created record copy in tmp_buf commits after tmp_buf is initialized.

        By OBVIOUS.
      - 2.2.1.3.2.1.3 Q.E.D.
    - 2.2.1.3.2.2 Q.E.D.
  - 2.2.1.3.3 If put creates a record copy in tmp_buf, there is no record copy in buf when put commits.

    By Lemma 3.
  - 2.2.1.3.4 2.2.1.3.2 and 2.2.1.3.3 contradicts.

    By OBVIOUS.
  - 2.2.1.3.5 Q.E.D.

- 2.2.1.4 Q.E.D.
- 2.2.2 Two record copies of k can not exist in the same location of a group.
  - 2.2.2.1 **CASE 1:** two record copies of k can't both exist in *data_array*
    - 2.2.2.1.1 Two record copies of k with the same key can't exist within one *data_array* when initialization.
      - 2.2.2.1.1.1 *data_array* is constructed using the record copies from *data_array* and *buf* of another group.

        By line 7, Algorithm 3; line 13, Algorithm 4.
      - 2.2.2.1.1.2 two record copies of k can't exist in *data_array* and *buf* of a group respectively.

        By 2.2.1.
      - 2.2.2.1.1.3 Q.E.D.
    - 2.2.2.1.2 *data_array* can't be inserted after initialization.

      By that data_array is implemented using fixed-length array.
    - 2.2.2.1.3 Q.E.D.
  - 2.2.2.2 **CASE 2**: two record copies of k can't both exist in *buf*

    By the property of delta index implementation that repeating call to `insert_buffer` only updates previous record copy's value.
  - 2.2.2.3 **CASE 3**: two record copies of k can't both exist in *tmp_buf*

    By the property of delta index implementation that repeating call to `insert_buffer` only updates previous record copy's value.
  - 2.2.2.4 Q.E.D.
- 2.2.3 Q.E.D.

- 2.3 Two record copies of k can not exist in two different groups.
  - 2.3.1 It suffices to prove that, during phase transition, two record copies of k can't exist in the old group and the new group respectively.
    - 2.3.1.1 Only during phase transition will there be two group visible (returned by `get_group`) to threads.

      By OBVIOUS.
    - 2.3.1.2 Q.E.D.
  - 2.3.2 During phase transition, two record copies of k can't exist in the old group and the new group respectively.
    - 2.3.2.1 If two record copies of k exist in the old group and the new group respectively, they must be created during phase transition.
      - 2.3.2.1.1 Before phase transition, there is one group contains k.

        By OBVIOUS.
      - 2.3.2.1.2 Two record copies of k can not exist within one group.

        By 2.
      - 2.3.2.1.3 Q.E.D.

- 2.3.2.2 If two record copies of k are created during phase transition, they are created by `put`.
  - 2.3.2.2.1 Only background threads and `put` creates new record copies.

    By OBVIOUS.
  - 2.3.2.2.2 Background thread does not create new record copy during phase transition.

    By line 9-10, Algorithm 3; 9-10, 16-17, Algorithm 4.
  - 2.3.2.2.3 Q.E.D.
- 2.3.2.3 `put` can not created two record copies of k in the old group and the new group during phase transition.
  - 2.3.2.3.1 **CASE 1**: compaction's phase transition & group_split's 2nd phase transition.
    - 2.3.2.3.1.1 If `put` creates two record copies of k in the old group and the new group during phase transition, two record copies will exist in both groups.
      - 2.3.2.3.1.1.1 All record copies are referenced in new group.

        By line 7-8, Algorithm 3; 13-14, Algorithm 4.
      - 2.3.2.3.1.1.2 Q.E.D.
    - 2.3.2.3.1.2 Two record copies of k can not exist within one group.

      By 2.2.
    - 2.3.2.3.1.3 Q.E.D.
  - 2.3.2.3.2 **CASE 2**: group_split's 1st phase transition.
    - 2.3.2.3.2.1 **CASE 1**: The duplicate record copy in new group is in tmp_buf.
      - 2.3.2.3.2.1.1 The duplicate record copy in old group is in data_array or buf.

        By OBVIOUS.
      - 2.3.2.3.2.1.2 The old group's data_array in buf are shared with the new group.

        By line 6, Algorithm 4.
      - 2.3.2.3.2.1.3 There are two record copies of k in the new group.

        By 2.3.2.3.2.1.1 and 2.3.2.3.2.1.2.
      - 2.3.2.3.2.1.4 There cannot be are two record copies of k in the new group.

        By 2.2.
      - 2.3.2.3.2.1.5 Q.E.D.
    - 2.3.2.3.2.2 **CASE 2**: The duplicate record copy in new group is in data_array or buf.
      - 2.3.2.3.2.2.1 two record copies will exist in both old and new groups.
        - 2.3.2.3.2.2.1.1 The old group's data_array in buf are shared with the new group.

          By line 6, Algorithm 4.
        - 2.3.2.3.2.2.1.2 Q.E.D.
      - 2.3.2.3.2.2.2 There cannot be are two record copies of k in the new group. By 2.2.
      - 2.3.2.3.2.2.3 Q.E.D.
    - 2.3.2.3.2.3 Q.E.D.
  - 2.3.2.3.3 Q.E.D.

        - 2.3.2.4 Q.E.D.
    - 2.3.3 Q.E.D.
- 2.4 Q.E.D.
- 3 Q.E.D.

Lemma 2. For any possible concurrent execution of XIndex, if a put or get operating on the key k commits at buf, then there is no record copy of k in data_array at the time the request commits.

Proof of Lemma 2.

- 1 **CASE 1**: the request is a put request.
    - 1.1 If put inserts/updates a record copy in buf, there is no record copy in data_array when put executes line 18 or 22.

      By the fact that otherwise put would then commit at line 25, Algorithm 2, instead of creating a record in buf.
    - 1.2 If there is no record copy in data_array when put executes line 18 or 22, there won't be a record copy in data_array after put executes line 18 or 22.
        - 1.2.1 *data_array* can't be inserted after initialization.

          By the implementation of data_array that provides no insert interface.
        - 1.2.2 put accesses buf after data_array is initialized.

          By OBVIOUS.
        - 1.2.3 Q.E.D.
    - 1.3 put commits after it executes line 18 or 22.

      By OBVIOUS.
    - 1.4 Q.E.D.
- 2 **CASE 2**: the request is a get request.
    - 2.1 If get commits at line 12, there is no record copy in data_array when get executes line 3, 5 or 7.

      By the fact that otherwise get would then commit at line 25, Algorithm 2.
    - 2.2 If there is no record copy in data_array when get executes line 5, there won't be a record copy in data_array after put executes line 3, 5 or 7.
        - 2.2.1 *data_array* can't be inserted after initialization.

          By the implementation of data_array that provides no insert interface.
        - 2.2.2 get accesses buf after data_array is initialized.

          By OBVIOUS.
        - 2.2.3 Q.E.D.
    - 2.3 get commits after it executes line 3, 5 or 7.

      By OBVIOUS.
    - 2.4 Q.E.D.
- 3 Q.E.D.

Lemma 3. For any possible concurrent execution of XIndex, if a put or get operating on the key k commits at tmp_buf (reads value from buf or insert/update value in tmp_buf), then there is no record copy of k in data_array or buf at the time the request commits.

Proof of Lemma 3.

- 1 **CASE 1**: the request is a put request.
  - 1.1 If put commits at line 34, there is no record copy in data_array when put executes line 18 or 22.

    By the fact that otherwise put would then commit at line 25, Algorithm 2, instead of creating a record in buf.
  - 1.2 If there is no record copy in data_array when put executes line 18 or 22, there won't be a record copy in data_array after put executes line 18 or 22.
    - 1.2.1 *data_array* can't be inserted after initialization.

      By the implementation of data_array that provides no insert interface.
    - 1.2.2 put accesses tmp_buf after data_array is initialized.

      By OBVIOUS.
    - 1.2.3 Q.E.D.
  - 1.3 If put commits at line 34, there is no record copy in buf when put executes line 28 or 32.

    By the fact that otherwise put would then commit at line 28 or 32, Algorithm 2, instead of creating a record in tmp_buf.
  - 1.4 If there is no record copy in data_array when put executes line 28 or 32, there won't be a record copy in data_array after put executes line 28 or 32.
    - 1.4.1 no put can insert new record copy to buf.
      - 1.4.1.1 buf is frozen before tmp_buf is initialized.

        By line 3-5, Algorithm 3; line 2-3, 7, Algorithm 4.
      - 1.4.1.2 tmp_buf is initialized.

        By the assumption that put will commit at line 34, Algorithm 2.
      - 1.4.1.3 Q.E.D.
    - 1.4.2 Q.E.D.
  - 1.5 put commits after it executes line 18, 22, 28 and 32.

    By OBVIOUS.
  - 1.6 Q.E.D.
- 2 **CASE 2**: the request is a get request.
  - 2.1 If get commits at line 12, there is no record copy in data_array when get executes line 3, 5 or 7.

    By the fact that otherwise get would then commit at line 5, Algorithm 2.
  - 2.2 If there is no record copy in data_array when get executes line 18 or 22, there won't be a record copy in data_array after get executes line 3, 5 or 7.
    - 2.2.1 *data_array* can't be inserted after initialization.

      By the implementation of data_array that provides no insert interface.

- 2.2.2 get accesses tmp_buf after data_array is initialized.

  By OBVIOUS.
- 2.2.3 Q.E.D.

- 2.3 If get commits at line 12, there is no record copy in buf when get executes line 10.

  By the fact that otherwise get would then commit at line 10, Algorithm 2.
- 2.4 If there is no record copy in data_array when get executes line 10, there won't be a record copy in data_array after get executes line 10.
  - 2.4.1 no put can insert new record copy to buf.
    - 2.4.1.1 buf is frozen before tmp_buf is initialized.

      By line 3-5, Algorithm 3; line 2-3, 7, Algorithm 4.
    - 2.4.1.2 tmp_buf is initialized.

      By the assumption that get will commit at line 12, Algorithm 2.
    - 2.4.1.3 Q.E.D.
  - 2.4.2 Q.E.D.
- 2.5 get commits after it executes line 3, 5, 7 and 10.

  By OBVIOUS.
- 2.6 Q.E.D.

- 3 Q.E.D.

Lemma 4. For any possible concurrent execution of XIndex, for any conflicting operations a and b, either a ≪ b or b ≪ a.

Proof of Lemma 4.

- 1 **CASE 1**: a and b are `puts`.
  - 1.1 It suffices to prove that any conflicting put a,b must not commit simultaneously 1) within one group; and 2) in two different groups.

    By OBVIOUS.
  - 1.2 any conflicting put a, b operating on key k can't commit simultaneously within one group.
    - 1.2.1 **CASE 1**: a, b must not commit at the same LP simultaneously.
      - 1.2.1.1 **CASE 1**: put commits at line 25, Algorithm 2.

        By atomicity of commit ensured by lock, line 21, Algorithm 2.
      - 1.2.1.2 **CASE 2**: put commits at line 28, 32 or 34, Algorithm 2.

        By the implementation of (temporary) delta index that ensures atomicity.
      - 1.2.1.3 Q.E.D.
    - 1.2.2 **CASE 2**: a, b must not commits at the different LP simultaneously.
      - 1.2.2.1 **CASE 1**: a commits at line 25, b commits at line 28 or 32 or 34.
        - 1.2.2.1.1 a record copy of k exists in *data_array* at the time a commits.

          By the fact that otherwise a will continue execution and commit after line 25, Algorithm 2.

- 1.2.2.1.2 no record copy of k exists in *data_array* at the time b commits.

  By Lemma 2, 3.

- 1.2.2.1.3 1.2.2.1.1 contradicts with 1.2.2.1.2.

  By OBVIOUS.

- 1.2.2.1.4 Q.E.D.

- 1.2.2.2 **CASE 2**: a commits at line 28, b commits line 32 or line 34.
  - 1.2.2.2.1 *buf_frozen* is false at the time a commits.

    By the fact that otherwise a would fail at branch condition at line 27 and cannot commit at line 28.

  - 1.2.2.2.2 all threads must have observed *buf_frozen* is true at the time b commits.
    - 1.2.2.2.2.1 *tmp_buf* have been initialized at the time b commits.

      By b commits at line 32, line 30-31 of Algorithm 2.

    - 1.2.2.2.2.2 *tmp_buf* is only initialized after all threads have observed *buf_frozen* is true.

      By line 3-5, Algorithm 2; line 2-3, 7, Algorithm 4.

    - 1.2.2.2.2.3 Q.E.D.
  - 1.2.2.2.3 1.2.2.2.1 contradicts with 1.2.2.2.2.

    By OBVIOUS.

  - 1.2.2.2.4 Q.E.D.
- 1.2.2.3 **CASE 3**: a commits at line 32, b commits at line 34.
  - 1.2.2.3.1 a record copy of k exists in *buf* at the time a commits.
    - 1.2.2.3.1.1 a record copy of k is successfully in-place updated in *buf.*

      By the fact that a commits at line 32.

    - 1.2.2.3.1.2 Q.E.D.
  - 1.2.2.3.2 no record copy of k exists in *buf* at the time b commits.

    By Lemma 3.

  - 1.2.2.3.3 1.2.2.3.1 contradicts with 1.2.2.3.2.

    By OBVIOUS.

  - 1.2.2.3.4 Q.E.D.
- 1.2.2.4 Q.E.D.

- 1.2.3 Q.E.D.

- 1.3 Any conflicting put a,b operating on key k must not commit simultaneously in two different groups.
  - 1.3.1 It suffices to prove that, during phase transition, two conflicting put a,b can't commit simultaneously in the old group and the new group.
    - 1.3.1.1 Only during phase transition will there be two group visible (returned by `get_group`) to threads.

      By OBVIOUS.

    - 1.3.1.2 Q.E.D.

- 1.3.2 During phase transition, two conflicting put a,b can't commit simultaneously in the old group and the new group respectively.
  - 1.3.2.1 **CASE 1**: compaction phase 1→phase 2.
    - 1.3.2.1.1 If a and b commit in the old group and the new group respectively, then there is a contradiction.
      - 1.3.2.1.1.1 b can only commit at line 25 or line 28.
        - 1.3.2.1.1.1.1 buf_frozen is false in the new group for all threads.

          By that group is created with buf_frozen is false.
        - 1.3.2.1.1.1.2 b can't commit at line 32 or line 34.

          By line 27, 29 of Algorithm 2.
        - 1.3.2.1.1.1.3 Q.E.D.
      - 1.3.2.1.1.2 **CASE 1**: b commits at line 25.
        - 1.3.2.1.1.2.1 a can only commit at line 25 or line 32 or line 34.
          - 1.3.2.1.1.2.1.1 All threads must have observed buf_frozen is true in the old group.

            By `mfence` and `rcu_barrier` and line 3-4 of Algorithm.
          - 1.3.2.1.1.2.1.2 b can't commit at line 28.

            By 1.3.2.1.1.2.1.1, line 27-28 of Algorithm 2.
          - 1.3.2.1.1.2.1.3 Q.E.D.
        - 1.3.2.1.1.2.2 **CASE 1**: a commits at line 25 or 32.
          - 1.3.2.1.1.2.2.1 a, b operate on the same record copy.
            - 1.3.2.1.1.2.2.1.1 There exists a record copy of k in data_array of the old group at the time a commits.

              By the fact that a commits at line 25 in the old group.
            - 1.3.2.1.1.2.2.2 There is a reference to the record copy at the time a commits in data_array of the new group .
              - 1.3.2.1.1.2.2.2.1 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

                By line 7, Algorithm 3.
              - 1.3.2.1.1.2.2.2.2 Q.E.D.
            - 1.3.2.1.1.2.2.3 b accesses the record copy in old group's data_array through the reference at the time b commits.
              - 1.3.2.1.1.2.2.3.1 If the record copy that b accesses is in buf, then Lemma 1 is violated.

                By OBVIOUS.
              - 1.3.2.1.1.2.2.3.2 Q.E.D.
            - 1.3.2.1.1.2.2.4 Q.E.D.
          - 1.3.2.1.1.2.2.2 a, b can't commit simultaneously.

            By line 21 of Algorithm 2 where lock is used to coordinate write accesses to the same record copy.

- 1.3.2.1.1.2.2.3 Q.E.D.
- 1.3.2.1.1.2.3 **CASE 2**: a commits at line 34.
  - 1.3.2.1.1.2.3.1 No record copy of k exists in *data_array* or *buf* of the old group at the time a commits.

    By Lemma 3.
  - 1.3.2.1.1.2.3.2 There exists a record copy of k in *data_array* or *buf* of the old group at the time b commits.
    - 1.3.2.1.1.2.3.2.1 There exists a reference in *data_array* of the new group that refers to a record copy of k.
      - 1.3.2.1.1.2.3.2.1.1 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

        By line 7 of Algorithm 3.
      - 1.3.2.1.1.2.3.2.1.2 Q.E.D.
    - 1.3.2.1.1.2.3.2.2 Q.E.D.
  - 1.3.2.1.1.2.3.3 1.3.2.1.1.2.3.1 contradicts with 1.3.2.1.1.2.3.2.

    By OBVIOUS.
  - 1.3.2.1.1.2.3.4 Q.E.D.
- 1.3.2.1.1.2.4 Q.E.D.

- 1.3.2.1.1.3 **CASE 2**: b commits at line 28.
  - 1.3.2.1.1.3.1 a can only commit at line 25 or line 32 or line 34.
    - 1.3.2.1.1.3.1.1 All threads must have observed buf_frozen is true in the old group.

      By `mfence` and `rcu_barrier` and line 3-4 of Algorithm.
    - 1.3.2.1.1.3.1.2 b can't commit at line 28.

      By 2.2.1.2.1.1, line 27-28 of Algorithm 2.
    - 1.3.2.1.1.3.1.3 Q.E.D.
  - 1.3.2.1.1.3.2 **CASE 1**: a commits at line 25, 32.
    - 1.3.2.1.1.3.2.1 There exist a record copy of k in data_array or buf of the old group.

      By a commits at data_array or buf of the old group.
    - 1.3.2.1.1.3.2.2 No record copy of k exists in data_array or buf of the old group.
      - 1.3.2.1.1.3.2.2.1 If there is a record copy of k exists in data_array or buf of the old group, then there is a reference in data_array of the new group.

        By line 7 of Algorithm 3.
      - 1.3.2.1.1.3.2.2.2 If there is a reference in data_array of the new group, b will not commit at line 28.

        By Lemma 2.
      - 1.3.2.1.1.3.2.2.3 Q.E.D.
    - 1.3.2.1.1.3.2.3 1.3.2.1.1.3.2.1 contradicts with 1.3.2.1.1.3.2.1.

      By OBVIOUS.

- 1.3.2.1.1.3.2.4 Q.E.D.
- 1.3.2.1.1.3.3 **CASE 2**: a commits at line 34.
  - 1.3.2.1.1.3.3.1 *buf* of the new group and *tmp_buf* of the old group refers to the same delta index instance.

    By line 8 of Algorithm 3.
  - 1.3.2.1.1.3.3.2 a,b can't commit simultaneously on the same delta index.

    By the property of delta index implementation.
  - 1.3.2.1.1.3.3.3 Q.E.D.
- 1.3.2.1.1.3.4 Q.E.D.
- 1.3.2.1.1.4 Q.E.D.
- 1.3.2.1.2 Q.E.D.

- 1.3.2.2 **CASE 2**: group split phase 1 → phase 2.
  - 1.3.2.2.1 **CASE 1**: a, b can't commit in *g_a*, *g_b* simultaneously.
    - 1.3.2.2.1.1 *g_a* and *g_b* are created to hava different key interval.

      By line 8 of Algorithm 4.
    - 1.3.2.2.1.2 If a,b commit at *g_a* and *g_b* respectively, then they are not conflicting.

      By 3.2.2.1.1.
    - 1.3.2.2.1.3 Q.E.D.
  - 1.3.2.2.2 **CASE 2**: a,b can't commit simultaneously in *g_a* (the new group) and the old group respectively.
    - 1.3.2.2.2.1 b can only commit at line 25.
      - 1.3.2.2.2.1.1 b see a frozen buf.

        By line 2-3, Algorithm 4.
      - 1.3.2.2.2.1.2 b do not see a initialized tmp_buf.

        By the fact that group_split does not initialize the old group's tmp_buf.
      - 1.3.2.2.2.1.3 Q.E.D.
    - 1.3.2.2.2.2 **CASE 1**: a commits at line 34.
      - 1.3.2.2.2.2.1 There is no record copy in data_array or buf in the old group.
        - 1.3.2.2.2.2.1.1 data_array and buf are shared between old group and new group.

          By line 6, Algorithm 4.
        - 1.3.2.2.2.2.1.2 If there is a record copy in data_array or buf in the new group, then a cannot commit at line 34.

          By Lemma 3.
      - 1.3.2.2.2.2.2 There is a record copy in data_array or buf in the old group.

        By the fact that b commits in the old group.
      - 1.3.2.2.2.2.3 1.3.2.2.2.2.1 and 1.3.2.2.2.2.2 contradicts.

        By OBVIOUS.
      - 1.3.2.2.2.2.4 Q.E.D.

- 1.3.2.2.2.3 **CASE 2**: a commits at line 25.
  - 1.3.2.2.2.3.1 a and b operate on the same record copy in `data_array`.

    By OBVIOUS.
  - 1.3.2.2.2.3.2 a and b cannot commit simultaneously.

    By the fact that a and b will be serialized by lock in line 21, Algorithm 2.
  - 1.3.2.2.2.3.3 Q.E.D.
- 1.3.2.2.2.4 **CASE 3**: a commits at line 28 or 32.
  - 1.3.2.2.2.4.1 There is no record copy in data_array in the old group.
    - 1.3.2.2.2.4.1.1 If there is a record copy in data_array in the old group, then a cannot commit at line 28 and 32.

      By Lemma 2.
    - 1.3.2.2.2.4.1.2 Q.E.D.
  - 1.3.2.2.2.4.2 There is a record copy in data_array in the old group.

    By the fact that b commit at line 25.
  - 1.3.2.2.2.4.3 1.3.2.2.2.4.1 and 1.3.2.2.2.4.2 contradict.

    By OBVIOUS.
  - 1.3.2.2.2.4.4 Q.E.D.
- 1.3.2.2.2.5 Q.E.D.

- 1.3.2.2.3 **CASE 3**: a,b can't commit in g_b, the old group simultaneously.

  By symmetry of b, and 1.3.2.2.2.
- 1.3.2.2.4 Q.E.D.

- 1.3.2.3 **CASE 3**: group split phase 2 → phase 3.
  - 1.3.2.3.1 $g\_a \rightarrow g\_a'$ and $g\_b \rightarrow g\_b'$ are two compaction process.

    By design of structure update.
  - 1.3.2.3.2 a,b can't commit simultaneously.

    By 1.3.2.1.
  - 1.3.2.3.3 Q.E.D.
- 1.3.2.4 Q.E.D.

- 1.3.3 Q.E.D.

- 1.4 Q.E.D.

- 2 **CASE 2**: a is put and b is get.
  - 2.1 It suffices to prove that any conflicting put a, get b can't commit simultaneously 1) within one group; and 2) in two different groups.

    By OBVIOUS.
  - 2.2 Any conflicting put a, get b operating on the same key k can't commit simultaneously within one groups.
    - 2.2.1 **CASE 1**: b commis at line 5 of Algorithm 2.
      - 2.2.1.1 **CASE 1**: a commits at line 25 of Algorithm 2.

        - 2.2.1.1.1 a,b can't commit simultaneously in *data_array*

          By that record copy in *data_array* is protected using OCC.
        - 2.2.1.1.2 Q.E.D.
    - 2.2.1.2 **CASE 2**: a commita at line 28 or line 32 or line 34 of Algorithm 2.
        - 2.2.1.2.1 No record copy of k exists in *data_array* at the time a commits.

          By the fact that a commits at *buf* or *tmp_buf* and lemma 2, 3.
        - 2.2.1.2.2 There exists a record copy of k in *data_array* at the time b commits.

          By b commits at *data_array*
        - 2.2.1.2.3 2.2.1.2.1 contradicts with 2.2.1.2.2.

          By OBVIOUS.
        - 2.2.1.2.4 Q.E.D.
    - 2.2.1.3 Q.E.D.
- 2.2.2 **CASE 2**: b commis at line 10 of Algorithm 2.
    - 2.2.2.1 **CASE 1**: a commits at line 25 of Algorithm 2.
        - 2.2.2.1.1 No record copy of k exists in *data_array* at the time b commits.

          By the fact that get b commit at *buf* and Lemma 2.
        - 2.2.2.1.2 There exists a record copy of k in *data_array* at the time a commits.

          By put a commits at *data_array*
        - 2.2.2.1.3 2.2.2.1.1 contradicts with 2.2.2.1.2.

          By OBVIOUS.
        - 2.2.2.1.4 Q.E.D.
    - 2.2.2.2 **CASE 2**: a commits at line 28,32 of Algorithm 2.
        - 2.2.2.2.1 a,b both commit at *buf*

          By a commits at line 28 or line 32, b commits at line 10.
        - 2.2.2.2.2 a,b can't commit simultaneously in the same *buf*

          By the property of delta index implementation.
        - 2.2.2.2.3 Q.E.D.
    - 2.2.2.3 **CASE 3**: a commits at line 34 of Algorithm 2.
        - 2.2.2.3.1 No record copy of k exists in *data_array* or *buf* at the time a commits.

          By the fact that a commits at line 34 and Lemma 3.
        - 2.2.2.3.2 There exists a record copy of k in *buf* at the time b commits.

          By b commits at *buf*
        - 2.2.2.3.3 2.2.2.3.1 contradicts with 2.2.2.3.2.

          By OBVIOUS.
        - 2.2.2.3.4 Q.E.D.
    - 2.2.2.4 Q.E.D.
- 2.2.3 **CASE 3**: b commits at line 12 of Algorithm 2.

- 2.2.3.1 **CASE 1**: a commit at line 25 of Algorithm 2.
  - 2.2.3.1.1 No record copy of k exists in *data_array* and *buf* at the time b commits.

    By the fact that that b commits at *tmp_buf* and lemma 3.
  - 2.2.3.1.2 There exists a record copy of k in *data_array* at the time a commits.

    By that a commits at *data_array*
  - 2.2.3.1.3 2.2.3.1.1 contradicts with 2.2.3.1.2.

    By OBVIOUS.
  - 2.2.3.1.4 Q.E.D.
- 2.2.3.2 **CASE 2**: a commits at line 28 of Algorithm 2.
  - 2.2.3.2.1 *tmp_buf* is initialized.

    By that b commits at *tmp_buf*
  - 2.2.3.2.2 *buf_frozen* is true for all threads.

    By `mfence` and `rcu_barrier` , line 3-5 of Algorithm 2.
  - 2.2.3.2.3 a can't commit at line 28.

    By line 27 of Algorithm 2, 2.2.3.2.2.
  - 2.2.3.2.4 Q.E.D.
- 2.2.3.3 **CASE 3**: a commits at line 32 of Algorithm 2.
  - 2.2.3.3.1 No record copy of k exists in *data_array* and *buf* at the time b commits.

    By the fact that b commits at *tmp_buf* and lemma 3.
  - 2.2.3.3.2 There exists a record copy of k in *buf*

    By a commits at line 32.
  - 2.2.3.3.3 2.2.3.3.1 contradicts with 2.2.3.3.2.

    By OBVIOUS.
  - 2.2.3.3.4 Q.E.D.
- 2.2.3.4 **CASE 3**: a commits at line 34 of Algorithm 2.
  - 2.2.3.4.1 a,b both commit at *tmp_buf*

    By that a commits at line 12 and b commits at line 34.
  - 2.2.3.4.2 a,b can't commit simultaneously.

    By the property of delta index implementation.
  - 2.2.3.4.3 Q.E.D.
- 2.2.3.5 Q.E.D.

- 2.2.4 Q.E.D.

- 2.3 Any conflicting put a, get b can't commit simultaneously on two groups.
  - 2.3.1 It suffices to prove that, during phase transition, two conflicting put a, get b can't commit simultaneously in the old group and the new group.
    - 2.3.1.1 Only during phase transition will there be two group visible (returned by `get_group`) to threads.

By OBVIOUS.

- 2.3.1.2 Q.E.D.

- 2.3.2 During phase transition, two conflicting put a, get b can't commit simultaneously in the old group and the new group respectively.
  - 2.3.2.1 **CASE 1**: compaction phase 1→phase 2.
    - 2.3.2.1.1 If a and b commit in the old group and the new group respectively, then there is a contradiction.
      - 2.3.2.1.1.1 **CASE 1**: put a commits in the old group, get b commits in the new group.
        - 2.3.2.1.1.1.1 get b can only commit at line 5 or line 10 in the new group.
          - 2.3.2.1.1.1.1.1 b can't commit at line 12 in the new group.

            By the fact that *tmp_buf* is not initialized in the new group, line 11 of Algorithm 2.

          - 2.3.2.1.1.1.1.2 Q.E.D.
        - 2.3.2.1.1.1.2 **CASE 1**:get b commits at line 5 of the new group.
          - 2.3.2.1.1.1.2.1 put a can only commit at line 25 or line 32 or line 34 in the old group.
            - 2.3.2.1.1.1.2.1.1 a can't commit at line 28 in the old group.
              - 2.3.2.1.1.1.2.1.1.1 buf_frozen is true for all threads in the old group.

                By mfence and rcu_barrier, line 3-4 of Algorithm 3.

              - 2.3.2.1.1.1.2.1.1.2 a can't insert *buf* in the old group.

                By 2.3.2.1.1.1.2.1.1.1, line 27-28 of Algorithm 2.

              - 2.3.2.1.1.1.2.1.1.3 Q.E.D.
            - 2.3.2.1.1.1.2.1.2 Q.E.D.
          - 2.3.2.1.1.1.2.2 **CASE 1**: a commits at line 25 in the old group.
            - 2.3.2.1.1.1.2.2.1 a,b both operate on the same record copy in *data_array* of the old group.
              - 2.3.2.1.1.1.2.2.1.1 There exists a record copy of k in *data_array* of the old group at the time a commits.

                By the fact that a commits in *data_array* of the old group.

              - 2.3.2.1.1.1.2.2.1.2 There is a reference to the record copy at the time a commits in *data_array* of the new group .
                - 2.3.2.1.1.1.2.2.1.2.1 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

                  By line 7 of Algorithm 3.

                - 2.3.2.1.1.1.2.2.1.2.2 Q.E.D.
              - 2.3.2.1.1.1.2.2.1.3 b accesses the same record copy in the old group's *data_array* through reference in the new group's *data_array*
                - 2.3.2.1.1.1.2.2.1.3.1 If the record copy that b accesses is in *buf* of the old group, then Lemma 1 is violated.

By OBVIOUS.

- 2.3.2.1.1.1.2.2.1.3.2 Q.E.D.

- 2.3.2.1.1.1.2.2.1.4 Q.E.D.

- 2.3.2.1.1.1.2.2.2 a,b can't commit simultaneously.

  By that record copies in *data_array* are protected with OCC.

- 2.3.2.1.1.1.2.2.3 Q.E.D.

- 2.3.2.1.1.1.2.3 **CASE 2**: a commits at line 32 or line 34 in the old group.

  - 2.3.2.1.1.1.2.3.1 There exists a record copy of k in *data_array* of the old group at the time a commits.

    By the fact that a commits in *data_array* of the old group.

  - 2.3.2.1.1.1.2.3.2 There is no record copy of k in *data_array* or *buf* of the old group at the time b commits.

    - 2.3.2.1.1.1.2.3.2.1 There is no reference to a record copy of k in *data_array* of the new group at the time b commits.

      By the fact that a commits at *buf* or *tmp_buf* of the new group, Lemma 2 and Lemma 3.

    - 2.3.2.1.1.1.2.3.2.2 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

      By line 7 of Algorithm 3.

    - 2.3.2.1.1.1.2.3.2.3 Q.E.D.

  - 2.3.2.1.1.1.2.3.3 2.3.2.1.1.1.2.3.1 contradicts with 2.3.2.1.1.1.2.3.2.

    By OBVIOUS.

  - 2.3.2.1.1.1.2.3.4 Q.E.D.

- 2.3.2.1.1.1.2.4 Q.E.D.

- 2.3.2.1.1.1.3 **CASE 2**: get b commits at line 10 of the new group.

  - 2.3.2.1.1.1.3.1 put a can only commit at line 25 or line 32 or line 34 in the old group.

    - 2.3.2.1.1.1.3.1.1 a can't commit at line 28 in the old group.

      - 2.3.2.1.1.1.3.1.1.1 buf_frozen is true for all threads in the old group.

        By mfence and rcu_barrier, line 3-4 of Algorithm 3.

      - 2.3.2.1.1.1.3.1.1.2 a can't insert *buf* in the old group.

        By 2.3.2.1.1.1.3.1.1.1, line 27-28 of Algorithm 2.

      - 2.3.2.1.1.1.3.1.1.3 Q.E.D.

    - 2.3.2.1.1.1.3.1.2 Q.E.D.

  - 2.3.2.1.1.1.3.2 **CASE 1**: a commits at line 25 in the old group.

    - 2.3.2.1.1.1.3.2.1 There exists a record copy of k in *data_array* of the old group at the time a commits.

      By that fact that a commits at *data_array* of the old group.

- 2.3.2.1.1.1.3.2.2 There is no record copy of k in *data_array* or *buf* of the old group at the time b commits.
  - 2.3.2.1.1.1.3.2.2.1 There is no referenece to a record copy of k in *data_array* of the new group.

    By that fact that b commits at *buf* of the new group, Lemma 2.
  - 2.3.2.1.1.1.3.2.2.2 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

    By line 7 of Algorithm 3.
  - 2.3.2.1.1.1.3.2.2.3 Q.E.D.
- 2.3.2.1.1.1.3.2.3 2.3.2.1.1.1.3.2.1 contradicts with 2.3.2.1.1.1.3.2.2.

  By OBVIOUS.
- 2.3.2.1.1.1.3.2.4 Q.E.D.

- 2.3.2.1.1.1.3.3 **CASE 2**: a commits at line 32 in the old group.
  - 2.3.2.1.1.1.3.3.1 There exists a record copy of k in *buf* of the old group at the time a commits.

    By that fact that a commits at *buf* of the old group and *buf* is in-place updated.
  - 2.3.2.1.1.1.3.3.2 There is no record copy of k in *data_array* or *buf* of the old group at the time b commits.
    - 2.3.2.1.1.1.3.3.2.1 There is no referenece to a record copy of k in *data_array* of the new group.

      By that fact that b commits at *buf* of the new group, Lemma 2.
    - 2.3.2.1.1.1.3.3.2.2 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

      By line 7 of Algorithm 3.
    - 2.3.2.1.1.1.3.3.2.3 Q.E.D.
  - 2.3.2.1.1.1.3.3.3 2.3.2.1.1.1.3.3.1 contradicts with 2.3.2.1.1.1.3.3.2.

    By OBVIOUS.
  - 2.3.2.1.1.1.3.3.4 Q.E.D.

- 2.3.2.1.1.1.3.4 **CASE 3**: a commits at line 34 in the old group.
  - 2.3.2.1.1.1.3.4.1 a commits at *tmp_buf* of the old group, b commits at *buf* of the new group.

    By the fact that a commits at line 34, b commits at line 10.
  - 2.3.2.1.1.1.3.4.2 *buf* of the new group and *tmp_buf* of the old group refers to the same delta index instance.

    By line 8 of Algorithm 3.
  - 2.3.2.1.1.1.3.4.3 a,b can't commit simultaneously on the same delta index.

    By the property of delta index implementation.
  - 2.3.2.1.1.1.3.4.4 Q.E.D.
- 2.3.2.1.1.1.3.5 Q.E.D.

- 2.3.2.1.1.1.4 Q.E.D.

- 2.3.2.1.1.2 **CASE 2**: put a commits in the new group, get b commits in the old group.
  - 2.3.2.1.1.2.1 put a can only commit at line 25 or line 28 in the new group.
    - 2.3.2.1.1.2.1.1 put a can't commit at line 32 or line 34 in the new group.

      By the fact that *buf_frozen* is false in the new group and line 29 ,31 of Algorithm 2.

    - 2.3.2.1.1.2.1.2 Q.E.D.
  - 2.3.2.1.1.2.2 **CASE 1**: put a commits at line 25 in the new group.
    - 2.3.2.1.1.2.2.1 **CASE 1**: get b commits at line 5 in the old group.
      - 2.3.2.1.1.2.2.1.1 a, b both operate on the same record copy in *data_array* of the old group.
        - 2.3.2.1.1.2.2.1.1.1 There exists a record copy of k in *data_array* of the old group at the time b commits.

          By the fact that b commits at *data_array*

        - 2.3.2.1.1.2.2.1.1.2 There exists a reference to the record copy in the new group's *data_array* at the time b commits.
          - 2.3.2.1.1.2.2.1.1.2.1 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

            By line 7 of Algorithm 3.

          - 2.3.2.1.1.2.2.1.1.2.2 Q.E.D.
        - 2.3.2.1.1.2.2.1.1.3 a accesses the record copy in old_group's data_array through the reference in *data_array* of the new group.
          - 2.3.2.1.1.2.2.1.1.3.1 If the reference points to a record copy in old_group's buf, then Lemma 1 is violated.

            By OBVIOUS.

          - 2.3.2.1.1.2.2.1.1.3.2 Q.E.D.
        - 2.3.2.1.1.2.2.1.1.4 Q.E.D.
      - 2.3.2.1.1.2.2.1.2 a, b can't commit simultaneously.

        By the fact that record copies in *data_array* is protected with OCC, 2.3.2.1.1.2.2.1.1.

      - 2.3.2.1.1.2.2.1.3 Q.E.D.
    - 2.3.2.1.1.2.2.2 **CASE 2**: get b commits at line 10 in the old group.
      - 2.3.2.1.1.2.2.2.1 a, b both operate on the same record copy in *buf* of the old group.
        - 2.3.2.1.1.2.2.2.1.1 There exists a record copy of k in *buf* of the old group at the time b commits.
          - 2.3.2.1.1.2.2.2.1.1.1 If there is no record copy of k in *buf* of the old group, then b will not commit at line 10, which introduces a contradiction.
            - 2.3.2.1.1.2.2.2.1.1.1.1 *get_from_buffer* at line 10 will return empty.

              By the assumption that no record copy of k in *buf* of the old group.

- 2.3.2.1.1.2.2.2.1.1.1.2 *tmp_buf* is intialized in the old group.

  By line 5 of Algorithm 3.

- 2.3.2.1.1.2.2.2.1.1.1.3 b will commit at line 12, which introduces a contradiction.

  By 2.3.2.1.1.2.2.2.1.1.1.1 and 2.3.2.1.1.2.2.2.1.1.1.2, line 11 of Algorithm 2.

- 2.3.2.1.1.2.2.2.1.1.1.4 Q.E.D.

- 2.3.2.1.1.2.2.2.1.1.2 Q.E.D.

- 2.3.2.1.1.2.2.2.1.2 There exists a reference to the record copy in *data_array* of the new group at the time b commits.

  - 2.3.2.1.1.2.2.2.1.2.1 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

    By line 7 of Algorithm 3.

  - 2.3.2.1.1.2.2.2.1.2.2 Q.E.D.

- 2.3.2.1.1.2.2.2.1.3 a accesses the record copy in old_group's buf through the reference in *data_array* of the new group.

  - 2.3.2.1.1.2.2.2.1.3.1 If the reference points to a record copy in old_group's data_array, then Lemma 1 is violated.

    By OBVIOUS.

  - 2.3.2.1.1.2.2.2.1.3.2 Q.E.D.

- 2.3.2.1.1.2.2.2.1.4 Q.E.D.

- 2.3.2.1.1.2.2.2.2 a, b can't commit simultaneously.

  By the property of delta index implementation, 2.3.2.1.1.2.2.2.1.

- 2.3.2.1.1.2.2.2.3 Q.E.D.

- 2.3.2.1.1.2.2.3 **CASE 3**: get b commits at line 12 in the old group.

  - 2.3.2.1.1.2.2.3.1 There is no record copy of k in *data_array* or *buf* of the old group at the time b commits.

    By the fact that b commits at *tmp_buf*, Lemma 3.

  - 2.3.2.1.1.2.2.3.2 There is a record copy of k in *data_array* or *buf* of the old group at the time a commits.

    - 2.3.2.1.1.2.2.3.2.1 There exists a reference in *data_array* of the new group that refers to a record copy of k at the time a commits.

      By the fact that a commits at *data_array* of the new group.

    - 2.3.2.1.1.2.2.3.2.2 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

      By line 7 of Algorithm 3.

    - 2.3.2.1.1.2.2.3.2.3 Q.E.D.

  - 2.3.2.1.1.2.2.3.3 2.3.2.1.1.2.2.3.1 contradicts with 2.3.2.1.1.2.2.3.2.

    By OBVIOUS.

- 2.3.2.1.1.2.2.3.4 Q.E.D.
- 2.3.2.1.1.2.2.4 Q.E.D.

- 2.3.2.1.1.2.3 **CASE 2**: put a commits at line 28 in the new group.
  - 2.3.2.1.1.2.3.1 **CASE 1**: get b commits at line 5 in the old group.
    - 2.3.2.1.1.2.3.1.1 There exist a record copy of k in *data_array* of the old group at the time b commits.

      By the fact that b commits at *data_array* of the old group.
    - 2.3.2.1.1.2.3.1.2 There is no record copy of k in *data_array* or *buf* of the old group at the time a commits.
      - 2.3.2.1.1.2.3.1.2.1 There is no record reference in *data_array* of the new group that refers to a record copy of k at the time a commits.

        By the fact that a commits at *buf* of the new group.
      - 2.3.2.1.1.2.3.1.2.2 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

        By line 7 of Algorithm 3.
      - 2.3.2.1.1.2.3.1.2.3 Q.E.D.
    - 2.3.2.1.1.2.3.1.3 2.3.2.1.1.2.3.1.1 contradicts with 2.3.2.1.1.2.3.1.2.

      By OBVIOUS.
    - 2.3.2.1.1.2.3.1.4 Q.E.D.
  - 2.3.2.1.1.2.3.2 **CASE 2**: get b commits at line 10 in the old group.
    - 2.3.2.1.1.2.3.2.1 There exist a record copy of k in *buf* of the old group at the time b commits.
      - 2.3.2.1.1.2.3.2.1.1 If there is no record copy of k in *buf* of the old group, then b will not commit at line 10, which introduces a contradiction.
        - 2.3.2.1.1.2.3.2.1.1.1 *get_from_buffer* at line 10 will return empty.

          By the assumption that no record copy of k in *buf* of the old group.
        - 2.3.2.1.1.2.3.2.1.1.2 *tmp_buf* is intialized in the old group.

          By line 5 of Algorithm 3.
        - 2.3.2.1.1.2.3.2.1.1.3 b will commit at line 12, which introduces a contradiction.

          By 2.3.2.1.1.2.3.2.1.1.1 and 2.3.2.1.1.2.3.2.1.1.2, line 11 of Algorithm 2.
        - 2.3.2.1.1.2.3.2.1.1.4 Q.E.D.
      - 2.3.2.1.1.2.3.2.1.2 Q.E.D.
    - 2.3.2.1.1.2.3.2.2 There is no record copy of k in *data_array* or *buf* of the old group at the time a commits.
      - 2.3.2.1.1.2.3.2.2.1 There is no record reference in *data_array* of the new group that refers to a record copy of k at the time a commits.

        By the fact that a commits at *buf* of the new group.

- 2.3.2.1.1.2.3.2.2.2 *data_array* of the new groups contains references of all the record copies in *data_array* and *buf* of the old group.

  By line 7 of Algorithm 3.

- 2.3.2.1.1.2.3.2.2.3 Q.E.D.

- 2.3.2.1.1.2.3.2.3 2.3.2.1.1.2.3.2.1 contradicts with 2.3.2.1.1.2.3.2.2.

  By OBVIOUS.

- 2.3.2.1.1.2.3.2.4 Q.E.D.

- 2.3.2.1.1.2.3.3 **CASE 3**: get b commits at line 12 in the old group.

  - 2.3.2.1.1.2.3.3.1 a commits at *buf* of the new group, b commits at *tmp_buf* of the old group.

    By the fact that a commits at line 28, b commits at line 12.

  - 2.3.2.1.1.2.3.3.2 *buf* of the new group and *tmp_buf* of the old group refers to the same delta index instance.

    By line 8 of Algorithm 3.

  - 2.3.2.1.1.2.3.3.3 a,b can't commit simultaneously on the same delta index.

    By the property of delta index implementation, 2.3.2.1.1.2.3.3.1, 2.3.2.1.1.2.3.3.2.

  - 2.3.2.1.1.2.3.3.4 Q.E.D.

- 2.3.2.1.1.2.3.4 Q.E.D.

- 2.3.2.1.1.2.4 Q.E.D.

- 2.3.2.1.1.3 Q.E.D.

- 2.3.2.1.2 Q.E.D.

- 2.3.2.2 **CASE 2**: group split phase 1 → phase 2.

  - 2.3.2.2.1 **CASE 1**: put a, get b can't commit in *g_a*, *g_b* simultaneously.

    - 2.3.2.2.1.1 *g_a* and *g_b* are created to hava different key interval.

      By line 8 of Algorithm 4.

    - 2.3.2.2.1.2 If a,b commit at *g_a* and *g_b* respectively, then they are not conflicting.

      By 2.3.2.2.1.1.

    - 2.3.2.2.1.3 Q.E.D.

  - 2.3.2.2.2 **CASE 2**: put a, get b can't commit simultaneously in *g_a* (the new group) and the old group respectively.

    - 2.3.2.2.2.1 **CASE 1**: put a commits in *g_a* (the new group), get b commits in the old group.

      - 2.3.2.2.2.1.1 b can only commit at line 5 or line 10 in the old group.

        - 2.3.2.2.2.1.1.1 *tmp_buf* is not initialized in the old group.

          By the fact that group_split does not initialize the old group's tmp_buf.

        - 2.3.2.2.2.1.1.2 b can't commit at line 12.

          By 2.3.2.2.2.1.1.1, line 11 of Algorithm 2.

- 2.3.2.2.2.1.1.3 Q.E.D.
- 2.3.2.2.2.1.2 **CASE 1**: b commits at line 5 in the old group.
  - 2.3.2.2.2.1.2.1 a can only commit at line 25 or line 32 or line 34 in *g_a*
    - 2.3.2.2.2.1.2.1.1 *buf_forzen* is true in *g_a*

      By line 8, line 10 of Algorithm 4.
    - 2.3.2.2.2.1.2.1.2 a cant' commit at line 28 in g_a.

      By 2.3.2.2.2.1.2.1.1, line 27 of Algorithm 2.
    - 2.3.2.2.2.1.2.1.3 Q.E.D.
  - 2.3.2.2.2.1.2.2 **CASE 1**: a commits at line 25 in *g_a*
    - 2.3.2.2.2.1.2.2.1 a commits at *data_array* of *g_a*, b commits a *data_array* of the old group.

      By the fact that a commits at line 25, b commits at line 5.
    - 2.3.2.2.2.1.2.2.2 a,b both operate on *data_array* of the old group.

      By the fact that *data_array* of *g_a* refers to *data_array* of the old group,3.2.2.2.1.3.1.1.
    - 2.3.2.2.2.1.2.2.3 a,b can't commit simultaneously in **the same *data_array*

      By the fact that record copise in *data_array* is protected by OCC, 2.3.2.2.2.1.2.2.2.
    - 2.3.2.2.2.1.2.2.4 Q.E.D.
  - 2.3.2.2.2.1.2.3 **CASE 2**: a commits at line 32 or line 34 in *g_a*
    - 2.3.2.2.2.1.2.3.1 There is no record copy of k in *data_array* of the old group at the time a commits.
      - 2.3.2.2.2.1.2.3.1.1 There is no record copy of k in *data_array* of *g_a* at the time a commits.

        By the fact that a commits at *buf* or *tmp_buf* of *g_a* , Lemma 2 and Lemma 3.
      - 2.3.2.2.2.1.2.3.1.2 *data_array* of *g_a* refers *data_array* of the old group.

        By line 6 of Algorithm 4.
      - 2.3.2.2.2.1.2.3.1.3 Q.E.D.
    - 2.3.2.2.2.1.2.3.2 There exist a record copy of k in *data_array* of the old group at the time b commits.

      By the fact that b commits at *data_array* of the old group.
    - 2.3.2.2.2.1.2.3.3 2.3.2.2.2.1.2.3.1 contradicts with 2.3.2.2.2.1.2.3.2.

      By OBVIOUS.
    - 2.3.2.2.2.1.2.3.4 Q.E.D.
  - 2.3.2.2.2.1.2.4 Q.E.D.
- 2.3.2.2.2.1.3 **CASE 2**: b commits at line 10 in the old group.
  - 2.3.2.2.2.1.3.1 a can only commit at line 25 or line 32 or line 34 in *g_a*
    - 2.3.2.2.2.1.3.1.1 *buf_forzen* is true in *g_a*

By line 8, line 10 of Algorithm 4.

- 2.3.2.2.2.1.3.1.2 a cant' commit at line 28 in g_a.

  By 2.3.2.2.2.1.3.1.1, line 27 of Algorithm 2.

- 2.3.2.2.2.1.3.1.3 Q.E.D.

- 2.3.2.2.2.1.3.2 **CASE 1**: a commits at line 25 in *g_a*
  - 2.3.2.2.2.1.3.2.1 There is no record copy of k in *data_array* of the old group at the time b commits.

    By the fact that b commits at *buf* of the old group, Lemma 2.

  - 2.3.2.2.2.1.3.2.2 There exists a record copy of k in *data_array* of the old group at the time a commits.
    - 2.3.2.2.2.1.3.2.2.1 There exists a record copy of k in *data_array* of *g_a* at the time a commits.

      By the fact that a commits at *data_array* of *g_a*

    - 2.3.2.2.2.1.3.2.2.2 *data_array* of *g_a* refers *data_array* of the old group.

      By line 6 of Algorithm 4.

    - 2.3.2.2.2.1.3.2.2.3 Q.E.D.
  - 2.3.2.2.2.1.3.2.3 2.3.2.2.2.1.3.2.1 contradicts with 2.3.2.2.2.1.3.2.2.

    By OBVIOUS.

  - 2.3.2.2.2.1.3.2.4 Q.E.D.

- 2.3.2.2.2.1.3.3 **CASE 2**: a commits at line 32 in *g_a*
  - 2.3.2.2.2.1.3.3.1 a commits at *buf* of *g_a*, b commits at *buf* of the old group.

    By the fact that a commits at lne 32, b commits at line 10.

  - 2.3.2.2.2.1.3.3.2 a,b commit at the same delta index instance.
    - 2.3.2.2.2.1.3.3.2.1 *buf* of *g_a* refers *buf* of the old group.

      By line 6, Algorithm 4.

    - 2.3.2.2.2.1.3.3.2.2 Q.E.D.
  - 2.3.2.2.2.1.3.3.3 a,b can't commit simultaneously on the same delta index.

    By the property of delta index implementataion, 2.3.2.2.2.1.3.3.2.

  - 2.3.2.2.2.1.3.3.4 Q.E.D.

- 2.3.2.2.2.1.3.4 **CASE 3**: a commits at line 34 in *g_a*
  - 2.3.2.2.2.1.3.4.1 There is no record copy of k in *data_array* or *buf* of the old group at the time a commits.
    - 2.3.2.2.2.1.3.4.1.1 There is no record copy of k in *data_array* or *buf* of the new group at the time a commits.

      By Lemma 3.

    - 2.3.2.2.2.1.3.4.1.2 *data_array and buf* are shared between **the new group and old group.

      By line 6, Algorithm 4.

- 2.3.2.2.2.1.3.4.1.3 Q.E.D.
- 2.3.2.2.2.1.3.4.2 There is a record copy of k in *buf* of the old group at the time b commits.

  By the fact that b commits at line 10.
- 2.3.2.2.2.1.3.4.3 2.3.2.2.2.1.3.4.1 contradicts with 2.3.2.2.2.1.3.4.2.

  By OBVIOUS.
- 2.3.2.2.2.1.3.4.4 Q.E.D.

- 2.3.2.2.2.1.3.5 Q.E.D.

- 2.3.2.2.2.1.4 Q.E.D.

- 2.3.2.2.2.2 **CASE 2**: put a commits in the old group, get b commits in *g_a(*the new group*)*
  - 2.3.2.2.2.2.1 can only commit at line 25 in the old group.
    - 2.3.2.2.2.2.1.1 a can't commit at line 28 in the old group.

      By the fact that *buf_frozen* is true for all threads in the old group, line 27 of Algorithm 2.
    - 2.3.2.2.2.2.1.2 a can't commit at line 32 or line 34 in the old group.

      By the fact that *tmp_buf* is not initialized in the old group, line 30-31 of Algorithm 2.
    - 2.3.2.2.2.2.1.3 Q.E.D.
  - 2.3.2.2.2.2.2 **CASE 1**: b commits at line 5 in *g_a*
    - 2.3.2.2.2.2.2.1 a commits at *data_array* of the old group, b commits at *data_array* of *g_a*

      By the fact that a commits at line 25, b commits at line 5.
    - 2.3.2.2.2.2.2.2 a,b both operate on *data_array* of the old group.

      By the fact that *data_array* of *g_a* refers to *data_array* of the old group, 2.3.2.2.2.2.2.2.
    - 2.2.3.2.2.2.2.2.3 a,b can't commit simultaneously in **the same *data_array*

      By the fact that record copise in *data_array* is protected by OCC, 2.3.2.2.2.2.2.2.
    - 2.2.3.2.2.2.2.2.4 Q.E.D.
  - 2.3.2.2.2.2.3 **CASE 2**: b commits at line 10 or line 12 in *g_a*
    - 2.3.2.2.2.2.3.1 There exists a record copy of k in *data_array* of the old group at the time a commits.

      By the fact that a commits at *data_array* of the old group.
    - 2.3.2.2.2.2.3.2 There is no record copy of k in *data_array* of the old group at the time b commits.
      - 2.3.2.2.2.2.3.2.1 There is no record copy of k in *data_array* of *g_a* at the time b commits.

        By the fact that b commits at *buf* or *tmp_buf*, Lemma 2 and Lemma 3.
      - 2.3.2.2.2.2.3.2.2 *data_array* of *g_a* refers to *data_array* of the old group.

By line 6 of Algorithm 4.

- 2.3.2.2.2.2.3.2.3 Q.E.D.

- 2.3.2.2.2.2.3.3 2.3.2.2.2.2.3.1 contradicts with 2.3.2.2.2.2.3.2.

  By OBVIOUS.

- 2.3.2.2.2.2.3.4 Q.E.D.

- 2.3.2.2.2.2.4 Q.E.D.

- 2.3.2.2.2.3 Q.E.D.

- 2.3.2.2.3 **CASE 3**: a,b can't commit in g_b, the old group simultaneously.

  By symmetry, proved by 2.3.2.2.2.

- 2.3.2.2.4 Q.E.D.

- 2.3.2.3 **CASE 3**: group split phase 2 → phase 3.

- 2.3.2.3.1 $g\_a \rightarrow g\_a'$ and $g\_b \rightarrow g\_b'$ are two compaction process.

  By design of structure update.

- 2.3.2.3.2 a,b can't commit simultaneously.

  By 2.3.2.1.

- 2.3.2.3.3 Q.E.D.

- 2.3.2.4 Q.E.D.

- 2.3.3 Q.E.D.

- 2.4 Q.E.D.

- 3 Q.E.D.

Lemma 5. (I2) For any possible concurrent execution of XIndex, for any key k, if there is one record copy of k in XIndex, then its value equals the value of the last committed `put` on k.

Proof of Lemma 5.

- 1 It suffices to prove that 1) if a put(k,v) commits, then there is a one and only one record copy of k whose value equals this `put`'s value; and 2) compaction and group_split do not make a record copy stale or lost.

- 1.1 The value of a record copy is changed by put, compaction and group_split.

  By OBVIOUS.

- 1.2 Q.E.D.

- 2 If a put(k,v) commits, then there is a one and only one record copy of k whose value equals this `put`'s value.

- 2.1 If a put(k,v) commits, then there is a record copy of k whose value equals this `put`'s value.

- 2.1.1 If there is no record copy of k in XIndex before put commits, then a new record copy of k will be created.

  By OBVIOUS.

- 2.1.2 If there is a record copy of k in XIndex before put commits, then its value will be successfully updated to the put's value.

- 2.1.2.1 put commits at either line 25, 28, 32 and 34, Algorithm 2.

By Definition 1.

- 2.1.2.2 **CASE 1**: put commits at line 25.
  - 2.1.2.2.1 There is a record copy in `data_array`.

    By OBVIOUS.
  - 2.1.2.2.2 The value of the record copy will be successfully updated to the put's value.

    By the use of lock that ensures atomicity and exclusive access to the record copy and the implementation of `update_record` that updates the record copy's value.
  - 2.1.2.2.3 Q.E.D.
- 2.1.2.3 **CASE 2**: put commits at line 28.
  - 2.1.2.3.1 There is a record copy in `buf`.

    By OBVIOUS.
  - 2.1.2.3.2 The value of the record copy will be successfully updated to the put's value.

    By the implementation of `insert_buffer` that ensures atomicity and updates the record copy's value.
  - 2.1.2.3.3 Q.E.D.
- 2.1.2.4 **CASE 3**: put commits at line 32.
  - 2.1.2.4.1 There is a record copy in `buf`.

    By OBVIOUS.
  - 2.1.2.4.2 The value of the record copy will be successfully updated to the put's value.

    By the implementation of `try_update_in_buffer` that ensures atomicity and updates the record copy's value.
  - 2.1.2.4.3 Q.E.D.
- 2.1.2.5 **CASE 4**: put commits at line 34.
  - 2.1.2.5.1 There is a record copy in `tmp_buf` .

    By OBVIOUS.
  - 2.1.2.5.2 The value of the record copy will be successfully updated to the put's value.

    By the implementation of `insert_buffer` that ensures atomicity and updates the record copy's value.
  - 2.1.2.5.3 Q.E.D.
- 2.1.2.6 Q.E.D.

- 2.1.3 Q.E.D.

- 2.2 If a put(k,v) commits, there is a one and only one record copy of k in XIndex.

  By Lemma 1.

- 2.3 Q.E.D.

- 3 compaction and group_split do not make a record copy stale or lost.
  - 3.1 During phase transition, a record copy will not be lost or stale.
    - 3.1.1 All record copies of the old group can be accessed in the new group.
      - 3.1.1.1 All record copies of the old group are referenced in the new group.

- 3.1.1.1.1 **CASE 1**: compaction's phase transition.
  - 3.1.1.1.1.1 Old group's record copies in data_array and buf are referenced in new group's data_array.

    By line 7, Algorithm 3.
  - 3.1.1.1.1.2 Old group's record copies in tmp_buf are referenced in new group's buf.

    By line 8, Algorithm 3.
  - 3.1.1.1.1.3 Q.E.D.
- 3.1.1.1.2 **CASE 2**: group_split's phase transition.
  - 3.1.1.1.2.1 **CASE 1**: group_split's first phase transition.
    - 3.1.1.1.2.1.1 Old group's record copies in data_array and buf are referenced in new group's data_array and buf.

      By line 6, Algorithm 4.
    - 3.1.1.1.2.1.2 Q.E.D.
  - 3.1.1.1.2.2 **CASE 2**: group_split's second phase transition.
    - 3.1.1.1.2.2.1 Old group's record copies in data_array and buf are referenced in new group's data_array.

      By line 13, Algorithm 4.
    - 3.1.1.1.2.2.2 Old group's record copies in tmp_buf are referenced in new group's buf.

      By line 14, Algorithm 4.
    - 3.1.1.1.2.2.3 Q.E.D.
  - 3.1.1.1.2.3 Q.E.D.
- 3.1.1.1.3 Q.E.D.
- 3.1.1.2 Q.E.D.
- 3.1.2 Modification to record copies can be observed in both old and new group.

  By 3.1.1.
- 3.1.3 Q.E.D.

- 3.2 Outside of phase transition, a record copy will not be lost or stale.
  - 3.2.1 compaction and group_split do not remove or update (change value) existing record copies.
    - 3.2.1.1 It suffices to prove that compaction's copy phase and group_split's third phase does not update existing record copies.
      - 3.2.1.1.1 Only compaction's copy phase and group_split's third phase update references in data_array to concrete value.

        By the design of out algorithm.
      - 3.2.1.1.2 Q.E.D.
    - 3.2.1.2 compaction's copy phase and group_split's third phase does not update (change value) existing record copies.

        - 3.2.1.2.1 Changing references in data_array to concrete values does not change record copies' value.
            - 3.2.1.2.1.1 References in data_array are atomically changed to concrete values during compaction's copy phase and group_split's third phase.

              By using lock to protect the change at line 18-22, Algorithm 3.
            - 3.2.1.2.1.2 The concrete value are copied to the place of previous reference.

              By line 19, Algorithm 3.
            - 3.2.1.2.1.3 Q.E.D.
        - 3.2.1.2.2 Q.E.D.
    - 3.2.1.3 Q.E.D.
  - 3.2.2 Q.E.D.
- 3.3 Q.E.D.
- 4 Q.E.D.

Lemma 6. (I3) For any possible concurrent execution of XIndex, for any key k, for `get` with key k, if there is one record copy of k in XIndex when `get` commits, then `get` returns the value of the record copy.

Proof of Lemma 6.

- 1 If a record copy exists, it is in either `data_array`, `buf` or `tmp_buf` of the group returned by `get_group`.
  - 1.1 **CASE 1**: during phase transition of compaction and group_split.
    - 1.1.1 During phase transition of compaction and group_split, if a record copy exists in old group, then it is referenced in the new group.

      By line 7-8, Algorithm 3; line 6, 13-14, Algorithm 4.
    - 1.1.2 Q.E.D.
  - 1.2 **CASE 2**: outside of phase transition of compaction and group_split.

    By OBVIOUS.
  - 1.3 Q.E.D.
- 2 **CASE 1**: record copy is in `data_array` when `get` commits.
  - 2.1 get can read the record copy from `data_array`.
    - 2.1.1 get will access `data_array` .

      By line 3-8, Algorithm 2.
    - 2.1.2 get can read the record copy from `data_array` .

      By the implementation of `read_record` that uses OCC to atomically read the record copy.
    - 2.1.3 Q.E.D.
  - 2.2 Q.E.D.
- 3 **CASE 2**: record copy is in `buf` when `get` commits.
  - 3.1 get reads the value of the record copy and commits at line 10, Algorithm 2.
    - 3.1.1 get will execute line 9-10, Algorithm 2.

- 3.1.1.1 get cannot read record copy from `data_array`.
  - 3.1.1.1.1 Otherwise get commits at line 5 and `data_array` then contains the record copy, which contradicts 3.

    By Lemma 1.
  - 3.1.1.1.2 Q.E.D.
- 3.1.1.2 Q.E.D.

- 3.1.2 When get executes line 9-10, Algorithm 2, it will read the value of the record copy.
  - 3.1.2.1 If get does not read the value of the record, then there is a contradiction.
    - 3.1.2.1.1 If get does not read the value of the record, it will check *tmp_buf*.

      By line 11, Algorithm 2.
    - 3.1.2.1.2 **CASE 1**: *tmp_buf* is not initialized.
      - 3.1.2.1.2.1 get will commit and returns empty at line 10, Algorithm 2.

        By Definition 1.
      - 3.1.2.1.2.2 buf does not contain record copy when get commit.

        By 3.1.2.1.2.1, Definition 1 and the implementation of the concurrent delta buffer that ensures `read_buffer` returns empty when there is no record when commits.
      - 3.1.2.1.2.3 3.1.2.1.2.1 contradicts 3.

        By OBVIOUS.
      - 3.1.2.1.2.4 Q.E.D.
    - 3.1.2.1.3 **CASE 2**: *tmp_buf* is initialized.
      - 3.1.2.1.3.1 *buf* does not contain record copy when get commit.
        - 3.1.2.1.3.1.1 get failed to read record from buf during execution.

          By 3.1.2.1.1.
        - 3.1.2.1.3.1.2 buf is frozen and cannot be inserted during get.
          - 3.1.2.1.3.1.2.1 tmp_buf is initialized after all threads observe frozen buf.

            By line 3-4, Algorithm 3; line 3-4, Algorithm 4.
          - 3.1.2.1.3.1.2.2 Q.E.D.
        - 3.1.2.1.3.1.3 Q.E.D.
      - 3.1.2.1.3.2 3.1.2.1.3.1 contradicts 3.

        By OBVIOUS.
      - 3.1.2.1.3.3 Q.E.D.
    - 3.1.2.1.4 Q.E.D.
  - 3.1.2.2 Q.E.D.
- 3.1.3 Q.E.D.

- 3.2 Q.E.D.

- 4 **CASE 3**: record copy is in `tmp_buf` when `get` commits.
  - 4.1 No record copy of k in `data_array` and `buf`.

By Lemma 3.

- 4.2 get reads the value of the record copy and commits at line 12, Algorithm 2.
  - 4.2.1 get will execute line 11-12, Algorithm 2.
    - 4.2.1.1 get will not get record from `data_array`
      - 4.2.1.1.1 Otherwise get commits at line 5 and `data_array` then contains the record copy, which contradicts 3.

        By OBVIOUS.
      - 4.2.1.1.2 Q.E.D.
    - 4.2.1.2 get will not get record from `buf`
      - 4.2.1.2.1 Otherwise get commits at line 10 and `buf` then contains the record copy, which contradicts 3.

        By OBVIOUS.
      - 4.2.1.2.2 Q.E.D.
    - 4.2.1.3 Q.E.D.
  - 4.2.2 When get executes line 11-12, Algorithm 2, it will read the value of the record copy.
    - 4.2.2.1 If get does not read the value of the record, then there is a contradiction.
      - 4.2.2.1.1 If get does not read the value of the record, get will commit and returns empty at line 12, Algorithm 2.

        By Definition 1.
      - 4.2.2.1.2 If get will commit and returns empty at line 12, Algorithm 2, there is no record copy in `tmp_buf` when get commits.

        By Definition 1 and the implementation of the concurrent delta buffer that ensures `read_buffer` returns empty when there is no record when commits.
      - 4.2.2.1.3 There is a record copy in `tmp_buf` when get commits.

        By 4.
      - 4.2.2.1.4 Q.E.D.
    - 4.2.2.2 Q.E.D.
  - 4.2.3 Q.E.D.
- 4.3 Q.E.D.

- 4 Q.E.D.

Lemma 7. For any possible concurrent execution of XIndex E, for any key k, for any committed get operation op_0 ∈ E, let S = {op | op ∈ E, op is committed `put` before op_0, op operates on k}, if $|S| \neq 0$, then $\exists$ op ∈ S: $\forall$ op' ∈ S/{op}, op' ≪ op.

Proof of Lemma 7.

- 1 If negation of Lemma 7 holds, then Lemma 4 is violated.
  - 1.1 If negation of Lemma 7 holds, then for some possible concurrent execution of XIndex E, for some key k, for some committed get operation op_0 ∈ E, let S = {op | op ∈ E, op is committed `put` before op_0, op operates on k}, if $|S| \neq 0$, then $\forall$ op ∈ S: op' ∈ S/{op}, op' ≪ op.

By OBVIOUS.

- 1.2 If for some possible concurrent execution of XIndex E, for some key k, for some committed get operation op_0 ∈ E, let S = {op | op ∈ E, op is committed `put` before op_0, op operates on k}, if |S| ≠ 0, then ∀ op ∈ S: ∃ op' ∈ S/{op}, op' ≪ op; then there is a circle consist of puts ∈ S, … ≪ op_i ≪ … ≪ op_j ≪ … ≪ op_i ≪ ….
  - 1.2.1 If such circle does not exist, then the latest committed put exists.

    By OBVIOUS.
  - 1.2.2 Q.E.D.
- 1.3 If such sequence exists, then op_i ≪ op_j and op_j ≪ op_i.

  By OBVIOUS.
- 1.4 If p_i ≪ p_j and p_j ≪ p_i, then Lemma 4 is violated.

  By OBVIOUS.
- 1.5 Q.E.D.

- 2 Q.E.D.

Definition 5. (latest committed `put`) For any possible concurrent execution of XIndex E, for any key k, for any committed get operation op_0 ∈ E, let S = {op | op ∈ E, op is committed `put` before op_0, op operates on k}. If |S| ≠ 0, we say op ∈ S is the latest committed `put` on k if ∀ op' ∈ S/{op}, op' ≪ op.

Theorem 1. (correctness condition) For any possible concurrent execution of XIndex E, for any key k, for `get` with key k, if there is `put` committed before `get` commits, then `get` returns the value of the last committed `put` on k before the `get`.

Proof of Theorem 1.

- 1 if there is `put` committed before `get` commits, then there is a record copy in XIndex.

  By Lemma 1.
- 2 If there is a record copy in XIndex, then get returns the value of the latest committed put.

  By Lemma 5, 6.
- 3 Q.E.D.

Definition 6. (sequential specification) XIndex `x` is an object supporting `put` and `get` operations. The state of XIndex is a set of records (key-value pairs) `x` = {<k0, v0>, ..., <kn, vn>} and is initially empty. The `put` and `get` operations induce the following state transitions of `x` with appropriate return values:

1. get(k), if there is a pair <ki, vi> exists in `x` that ki=k, returns v; else returns empty; `x` always remains unchanged;
2. put(k, v), if there is a pair <ki, vi> exists in `x` that ki=k, changes `x` to `x` \ {<ki, vi>} ∩ {<ki, v>}; else changes `x` to `x` ∩ {<k, v>}; put has no return value;

Theorem 2. (linearizability) For any possible concurrent execution of XIndex E, for any key k, the following holds:

- operations on k can be serialized by the linearization point; and.
- for operations a and b, if a finishes before b starts, then a is serialized before b.

Proof of Theorem 2.

- 1 Operations on k can be serialized by the linearization point.
  - 1.1 It suffices to prove that conflicting operations on k can be serialized by the linearization point.

    By OBVIOUS.
  - 1.2 Conflicting operations on k can be serialized by the linearization point.
    - 1.2.1 For conflicting operations a and b, a is either serialized before or after b.
      - 1.2.1.1 Conflicting operations a and b, either a ≪ b or b ≪ a.

        By Lemma 4.
      - 1.2.1.2 Q.E.D.
    - 1.2.2 The value get returns in the serialized execution is the same as its return value in the concurrent execution.
      - 1.2.2.1 **CASE 1**: get returns empty in the serialized execution.
        - 1.2.2.1.1 No put is serialized before get.

          By the Definition 6.
        - 1.2.2.1.2 If no put is serialized before get, then no put commits before get.

          By the fact that operations are serialized according to linearization point.
        - 1.2.2.1.3 If no put commits before get, get returns empty in concurrent execution.

          By OBVIOUS.
        - 1.2.2.1.4 Q.E.D.
      - 1.2.2.2 **CASE 2**: get returns non-empty in the serialized execution.
        - 1.2.2.2.1 get returns the value of the put serialized nearest before the get.

          By the Definition 6.
        - 1.2.2.2.2 If a put is serialized nearest before the get, then the put is the last committed put before the get.

          By the fact that operations are serialized according to linearization point and Definition 5.
        - 1.2.2.2.3 If there is `put` committed before `get` commits, get returns the value of the last committed `put` on k before the `get` in concurrent execution.

          By Theorem 1.
        - 1.2.2.2.4 Q.E.D.
      - 1.2.2.3 Q.E.D.
    - 1.2.3 Q.E.D.
  - 1.3 Q.E.D.
- 2 For operations a and b, if a finishes before b starts, then a is serialized before b.
  - 2.1 It suffices to prove that, for conflicting operations a and b, if a finishes before b starts, then a is serialized before b.

    By OBVIOUS.
  - 2.2 For conflicting operations a and b, if a finishes before b starts, then a is serialized before b.
    - 2.2.1 If a finishes before b starts, then a ≪ b.

- 2.2.1.1 a and b both commit after operation starts and before operation ends.

  By Definition 1.
- 2.2.1.2 Q.E.D.

- 2.2.2 If a ≪ b, then a is serialized before b.

  By OBVIOUS.
- 2.2.3 Q.E.D.

- 2.3 Q.E.D.

- 3 Q.E.D.