DB Final Project

Team 15
104062203 陳涵宇
104062232 廖子毅
104062234 林士軒

Outline

- Paper introduction
    - Motivation and Problem
    - Main Idea
        - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

Outline

- Paper introduction
  - Motivation and Problem
  - Main Idea
  - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

On Optimistic Methods for Concurrency Control

ACM Transactions on Database System
June 1981
Citation count: 1678

H.T. KUNG and JOHN t. ROBINSON
Carnegie-Mellon University

Motivation and Problem

- Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism.

- Applications for which nonblocking concurrency controls should be more efficient than locking are discussed.

- Assumption: multiple transactions can frequently complete without interfering with each other.

Outline

- Paper introduction
  - Motivation and Problem
  - Main Idea
  - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

Main Idea

- divided transaction into three phases: read phase, validation phase, write phase

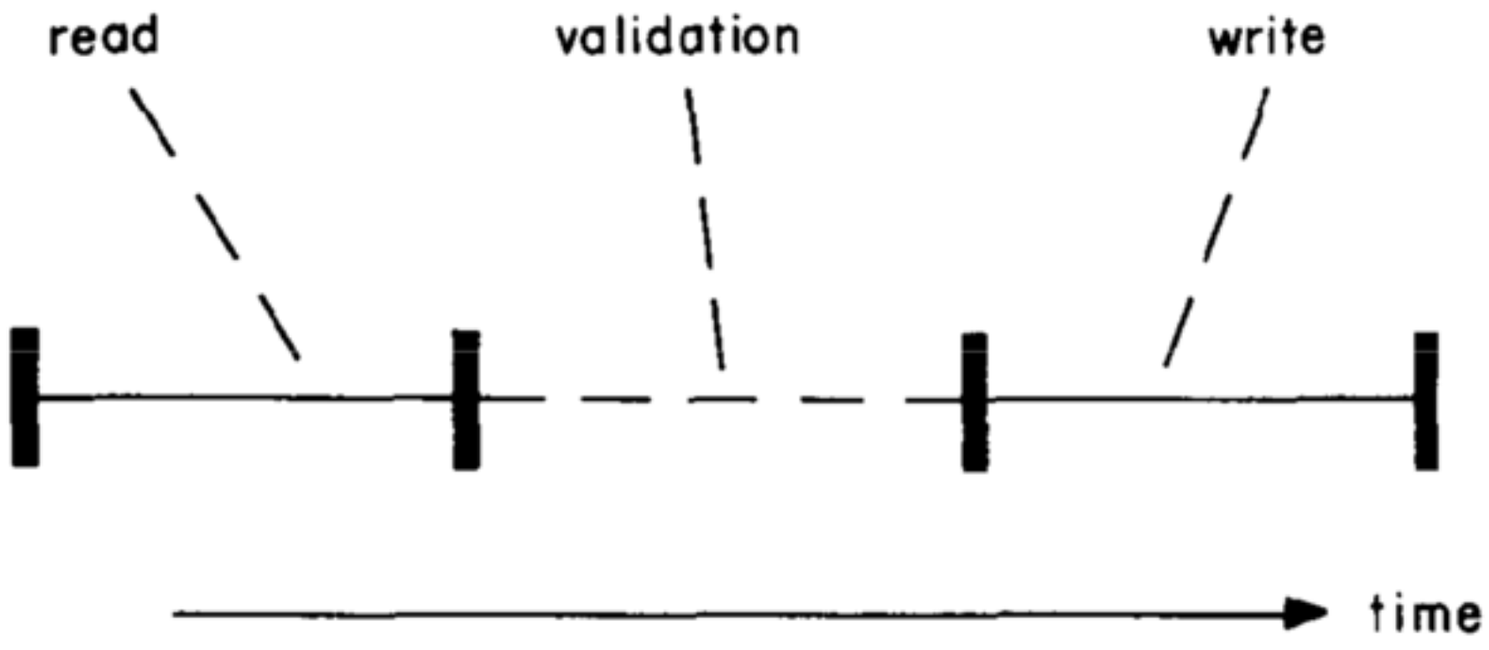


Fig. 1. The three phases of a transaction.

Main Idea

- read phase: all writes take place on local copies of the nodes to be modified. create and maintain four sets.

```
tcreate = (
   n := create;
   create set := create set ∪ {n};
   return n)

twrite(n, i, v) = (
   if n ∈ create set
        then write(n, i, v)
   else if n ∈ write set
        then write(copies[n], i, v)
   else (
        m := copy(n);
        copies[n] := m;
        write set := write set ∪ {n};
        write(copies[n], i, v)))

tread(n, i) = (
   read set := read set ∪ {n};
   if n ∈ write set
        then return read(copies[n], i)
   else
        return read(n, i))

tdelete(n) = (
   delete set := delete set ∪ {n}).
```

Main Idea

- validation phase: the changes of the transaction made will not cause a loss of integrity.

(1) T_i completes its write phase before T_j starts its read phase.
(2) The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.
(3) The write set of T_i does not intersect the read set *or* the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

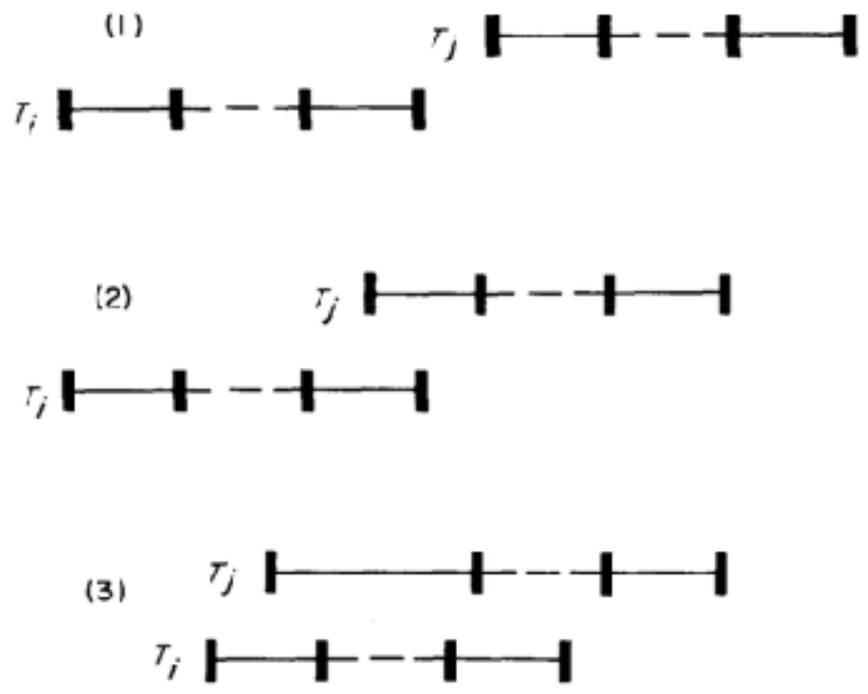


Fig. 2. Possible interleaving of two transactions.

Main Idea

- serial validation

tbegin = (
create set := *empty*;
read set := *empty*;
write set := *empty*;
delete set := *empty*;
start tn := *tnc*)

Main Idea

- serial validation

```
tend = (
⟨finish tn := tnc;
 valid := true;
 for t from start tn + 1 to finish tn do
      if (write set of transaction with transaction number t intersects read set)
         then valid := false;
 if valid
      then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
 if valid
      then (cleanup)
      else (backup)).
```

Main Idea

- parallel validation

```
tend = (
   ⟨finish tn := tnc;
    finish active := (make a copy of active);
    active := active ∪ {id of this transaction}⟩;
   valid := true:
for t from start tn + 1 to finish tn do
      if (write set of transaction with transaction number t intersects read set)
         then valid := false;
for i ∈ finish active do
      if (write set of transaction Ti intersects read set or write set)
         then valid := false;
if valid
      then (
         (write phase);
         ⟨tnc := tnc + 1;
          tn := tnc;
          active := active—{id of this transaction}⟩;
         (cleanup))
      else (
         ⟨active := active—{id of transaction}⟩;
         (backup)).
```

Main Idea

- write phase: the local copies are made global

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Outline

- Paper introduction
  - Motivation and Problem
  - Main Idea
  - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

Conclusion

- pros
  - reach high performance in low conflict rate
    - without maintain lock protocol overhead
- cons
  - the DBMS may has extreme low throughput when high conflict rate

Outline

- Paper introduction
  - Motivation and Problem
  - Main Idea
  - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

pre-work

- create three per-transaction sets (write, read, delete)

```
public Transaction(TransactionMgr txMgr, TransactionLifecycleListener concurMgr,
        TransactionLifecycleListener recoveryMgr, TransactionLifecycleListener bufferMgr, boolean readOnly,
    long txNum, long startTn) {
this.concurMgr = (ConcurrencyMgr) concurMgr;
this.recoveryMgr = (RecoveryMgr) recoveryMgr;
this.bufferMgr = (BufferMgr) bufferMgr;
this.txNum = txNum;
this.startTn = startTn;
this.readOnly = readOnly;
this.writeSet = new HashMap<RecordField, Constant>();
this.readSet = new HashSet<RecordField>();
this.deleteSet = new HashSet<RecordField>();
this.cCertified = false;
this.rCertified = false;
```

```
public class RecordField {
    public String tblName;
    public RecordId rid;
    public String fldName;
```

pre-work

- create write sets and tnc in txMgr for success committed tx

- create active set using in validation

```
public class TransactionMgr implements TransactionLifecycleListener {
    private long nextTxNum = 0;
    private long tnc = 0;
    // Optimization: Use separate lock for nextTxNum
    private Object txNumLock = new Object();
    private Object tncLock = new Object();
    private HashMap<Long, HashSet<RecordField>> writeSets;
    private HashSet<Transaction> activeSets;
    private Object activeLock = new Object();
```

Read Phase

- four function map to four action
  - insert() : create + write
  - setVal() : write
  - getVal() : read
  - delete() : delete

insert()

- create + write

```
tcreate = (
    n := create;
    create set := create set ∪ {n};
    return n)

twrite(n, i, v) = (
    if n ∈ create set
        then write(n, i, v)
    else if n ∈ write set
        then write(copies[n], i, v)
    else (
        m := copy(n);
        copies[n] := m;
        write set := write set ∪ {n};
        write(copies[n], i, v)))
```

setVal()

$$
\begin{aligned}
&twrite(n, i, v) = (\\
&\quad \textbf{if } n \in create\ set\\
&\qquad \textbf{then } write(n, i, v)\\
&\quad \textbf{else if } n \in write\ set\\
&\qquad \textbf{then } write(copies[n], i, v)\\
&\quad \textbf{else } (\\
&\qquad m := copy(n);\\
&\qquad copies[n] := m;\\
&\qquad write\ set := write\ set \cup \{n\};\\
&\qquad write(copies[n], i, v)))
\end{aligned}
$$

RecordFile

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);
    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.rollbackCertified() && !tx.commitCertified() && !isTempTable()) {
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```

modify in tx write set

Transaction

```
public void putVal(String tblName, RecordId rid, String fldName, Constant val) {
    writeSet.put(new RecordField(tblName, rid, fldName), val);
}
```

setVal()

$$
\begin{aligned}
&twrite(n, i, v) = (\\
&\quad \textbf{if } n \in create\ set\\
&\qquad \textbf{then } write(n, i, v)\\
&\quad \textbf{else if } n \in write\ set\\
&\qquad \textbf{then } write(copies[n], i, v)\\
&\quad \textbf{else } (\\
&\qquad m := copy(n);\\
&\qquad copies[n] := m;\\
&\qquad write\ set := write\ set \cup \{n\};\\
&\qquad write(copies[n], i, v)))
\end{aligned}
$$

RecordFile

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);
    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.rollbackCertified() && !tx.commitCertified() && !isTempTable()) {
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```

it's not rollback or commit yet

Transaction

```
public void putVal(String tblName, RecordId rid, String fldName, Constant val) {
    writeSet.put(new RecordField(tblName, rid, fldName), val);
}
```

getVal()

$tread(n, i) = ($
  $read\ set := read\ set \cup \{n\};$
  if $n \in write\ set$
    then return $read(copies[n], i)$
else
    return $read(n, i))$

RecordFile

```
public Constant getVal(String fldName) {
    Constant val;
    val = tx.getVal(ti.tableName(), currentRecordId(), fldName);
    if (val != null)
        return val;
    return rp.getVal(fldName);
}
```

check if n is in write set

Transaction

```
public Constant getVal(String tblName, RecordId rid, String fldName) {
    readSet.add(new RecordField(tblName, rid, fldName));
    return writeSet.get(new RecordField(tblName, rid, fldName));
}
```

getVal()

$$
\begin{aligned}
&tread(n, i) = (\\
&\quad read\ set := read\ set \cup \{n\};\\
&\quad \textbf{if } n \in write\ set\\
&\qquad \textbf{then return } read(copies[n], i)\\
&\textbf{else}\\
&\qquad \textbf{return } read(n, i))
\end{aligned}
$$

RecordFile

```
public Constant getVal(String fldName) {
    Constant val;
    val = tx.getVal(ti.tableName(), currentRecordId(), fldName);
    if (val != null)
        return val;
    return rp.getVal(fldName);
}
```

Transaction

```
public Constant getVal(String tblName, RecordId rid, String fldName) {
    readSet.add(new RecordField(tblName, rid, fldName));
    return writeSet.get(new RecordField(tblName, rid, fldName));
}
```

add n into read set

delete()

$$tdelete(n) = ($$
$$\quad delete\ set := delete\ set \cup \{n\}).$$

Transaction

```
public void delete(String tblName, RecordId rid) {
    deleteSet.add(new RecordField(tblName, rid, ""));
}
```

RecordFile

```
public void delete() {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();

    if(!tx.rollbackCertified() && !tx.commitCertified()) {
        tx.delete(ti.tableName(), currentRecordId());
    } else {

        if (fhp == null)
            fhp = openHeaderForModification();

        // Log that this logical operation starts
        RecordId deletedRid = currentRecordId();
        tx.recoveryMgr().logLogicalStart();

        // Delete the current record
        rp.delete(fhp.getLastDeletedSlot());
        fhp.setLastDeletedSlot(currentRecordId());

        // Log that this logical operation ends
        tx.recoveryMgr().logRecordFileDeletionEnd(ti.tableName(),
                deletedRid.block().number(), deletedRid.id());

        // Close the header (release the header lock)
        closeHeader();

    }
}
```

modify in tx delete set

delete()

$$tdelete(n) = ($$
$$\quad delete\ set := delete\ set \cup \{n\}).$$

Transaction

```
public void delete(String tblName, RecordId rid) {
    deleteSet.add(new RecordField(tblName, rid, ""));
}
```

add n into delete set

RecordFile

```
public void delete() {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();

    if(!tx.rollbackCertified() && !tx.commitCertified()) {
        tx.delete(ti.tableName(), currentRecordId());
    } else {

        if (fhp == null)
            fhp = openHeaderForModification();

        // Log that this logical operation starts
        RecordId deletedRid = currentRecordId();
        tx.recoveryMgr().logLogicalStart();

        // Delete the current record
        rp.delete(fhp.getLastDeletedSlot());
        fhp.setLastDeletedSlot(currentRecordId());

        // Log that this logical operation ends
        tx.recoveryMgr().logRecordFileDeletionEnd(ti.tableName(),
                deletedRid.block().number(), deletedRid.id());

        // Close the header (release the header lock)
        closeHeader();

    }
}
```

Validation Phase

- implement parallel version

```
tend = (
   ⟨finish tn := tnc;
    finish active := (make a copy of active);
    active := active ∪ {id of this transaction}⟩;
   valid := true:
for t from start tn + 1 to finish tn do
      if (write set of transaction with transaction number t intersects read set)
         then valid := false;
for i ∈ finish active do
      if (write set of transaction Ti intersects read set or write set)
         then valid := false;
if valid
      then (
         (write phase);
         ⟨tnc := tnc + 1;
          tn := tnc;
          active := active—{id of this transaction}⟩;
         (cleanup))
      else (
         ⟨active := active—{id of transaction}⟩;
         (backup)).
```

tbegin

$$
\begin{aligned}
&tbegin = (\\
&\quad create\ set := empty;\\
&\quad read\ set := empty;\\
&\quad write\ set := empty;\\
&\quad delete\ set := empty;\\
&\quad start\ tn := tnc\ )
\end{aligned}
$$

TransactionMgr

```
private Transaction createTransaction(int isolationLevel, boolean readOnly, long txNum) {
        tx = new Transaction(this, concurMgr, recoveryMgr, bufferMgr, readOnly, txNum, tnc);
```

pass tnc to Transaction constructor

Transaction

```
public Transaction(TransactionMgr txMgr, TransactionLifecycleListener concurMgr,
        TransactionLifecycleListener recoveryMgr, TransactionLifecycleListener bufferMgr, boolean readOnly,
        long txNum, long startTn) {
    this.concurMgr = (ConcurrencyMgr) concurMgr;
    this.recoveryMgr = (RecoveryMgr) recoveryMgr;
    this.bufferMgr = (BufferMgr) bufferMgr;
    this.txNum = txNum;
    this.startTn = startTn;
    this.readOnly = readOnly;
    this.writeSet = new HashMap<RecordField, Constant>();
    this.readSet = new HashSet<RecordField>();
    this.deleteSet = new HashSet<RecordField>();
    this.cCertified = false;
    this.rCertified = false;
```

tend

```
tend = (
   ⟨finish tn := tnc;
    finish active := (make a copy of active);
    active := active ∪ {id of this transaction}⟩;
    valid := true;
 for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
          then valid := false;
for i ∈ finish active do
       if (write set of transaction T_i intersects read set or write set)
          then valid := false;
if valid
       then (
          (write phase);
          ⟨tnc := tnc + 1;
           tn := tnc;
           active := active—{id of this transaction}⟩;
          (cleanup))
       else (
          ⟨active := active—{id of transaction}⟩;
          (backup))).
```

Transaction

validate before write

```
public void commit() {

    if(VanillaDb.txMgr().validate(this)) {
        commitCertify();
        commitWorkspace();
        for (TransactionLifecycleListener l : lifecycleListeners)
            l.onTxCommit(this);

        if (logger.isLoggable(Level.FINE))
            logger.fine("transaction " + txNum + " committed");

        VanillaDb.txMgr().finishedCommit(writeSet.keySet());
        VanillaDb.txMgr().endCommit(this);
    } else {
        VanillaDb.txMgr().endCommit(this);
        throw new ValidAbortException("abort tx." + this.txNum + " for no validation");
    }

}
```

validate()

TransactionMgr

```
public boolean validate(Transaction tx) {
    long start = tx.getStartTn();
    long finish = 0;
    HashSet<Transaction> active;
    synchronized (tncLock) {
        synchronized (activeLock) {
            finish = tnc;
            active = new HashSet<Transaction>(activeSets);
            activeSets.add(tx);
        }
    }
    HashSet<RecordField> readSet = tx.getReadSet();
    HashSet<RecordField> writeSet = tx.getWriteSet();

    // write set of transaction from start tn+1 to finish tn interacts read set
    for(long it=start+1; it<=finish; ++it) {
        HashSet<RecordField> hs = new HashSet<RecordField>(writeSets.get(it));
        if(hs.retainAll(readSet)) {
            return false;
        }
    }

    // write set of transaction in active Set interacts read set or write set
    for(Transaction activeTx: active) {
        HashSet<RecordField> hs1 = new HashSet<RecordField>(activeTx.getWriteSet());
        HashSet<RecordField> hs2 = new HashSet<RecordField>(hs1);
        if(hs1.retainAll(readSet)) {
            return false;
        }
        if(hs2.retainAll(writeSet)) {
            return false;
        }
    }

    return true;
}
```

make copies of tnc and active tx

validate()

TransactionMgr

```
public boolean validate(Transaction tx) {
    long start = tx.getStartTn();
    long finish = 0;
    HashSet<Transaction> active;
    synchronized (tncLock) {
        synchronized (activeLock) {
            finish = tnc;
            active = new HashSet<Transaction>(activeSets);
            activeSets.add(tx);
        }
    }
    HashSet<RecordField> readSet = tx.getReadSet();
    HashSet<RecordField> writeSet = tx.getWriteSet();

    // write set of transaction from start tn+1 to finish tn interacts read set
    for(long it=start+1; it<=finish; ++it) {
        HashSet<RecordField> hs = new HashSet<RecordField>(writeSets.get(it));
        if(hs.retainAll(readSet)) {
            return false;
        }
    }

    // write set of transaction in active Set interacts read set or write set
    for(Transaction activeTx: active) {
        HashSet<RecordField> hs1 = new HashSet<RecordField>(activeTx.getWriteSet());
        HashSet<RecordField> hs2 = new HashSet<RecordField>(hs1);
        if(hs1.retainAll(readSet)) {
            return false;
        }
        if(hs2.retainAll(writeSet)) {
            return false;
        }
    }

    return true;
}
```

follow condition (2)

(2) The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.

validate()

TransactionMgr

```
public boolean validate(Transaction tx) {
    long start = tx.getStartTn();
    long finish = 0;
    HashSet<Transaction> active;
    synchronized (tncLock) {
        synchronized (activeLock) {
            finish = tnc;
            active = new HashSet<Transaction>(activeSets);
            activeSets.add(tx);
        }
    }
    HashSet<RecordField> readSet = tx.getReadSet();
    HashSet<RecordField> writeSet = tx.getWriteSet();

    // write set of transaction from start tn+1 to finish tn interacts read set
    for(long it=start+1; it<=finish; ++it) {
        HashSet<RecordField> hs = new HashSet<RecordField>(writeSets.get(it));
        if(hs.retainAll(readSet)) {
            return false;
        }
    }

    // write set of transaction in active Set interacts read set or write set
    for(Transaction activeTx: active) {
        HashSet<RecordField> hs1 = new HashSet<RecordField>(activeTx.getWriteSet());
        HashSet<RecordField> hs2 = new HashSet<RecordField>(hs1);
        if(hs1.retainAll(readSet)) {
            return false;
        }
        if(hs2.retainAll(writeSet)) {
            return false;
        }
    }

    return true;
}
```

follow condition (3)

(3) The write set of T_i does not intersect the read set *or* the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

end validation

tend = (
  ⟨*finish tn* := *tnc*;
   finish active := (*make a copy of active*);
   active := *active* ∪ {*id of this transaction*}⟩;
   valid := **true**;
for *t* **from** *start tn* + 1 **to** *finish tn* **do**
    if (*write set of transaction with transaction number t intersects read set*)
      then *valid* := **false**;
for *i* ∈ *finish active* **do**
    if (*write set of transaction* T_i *intersects read set or write set*)
      then *valid* := **false**;
if *valid*
    then (
      (*write phase*);
      ⟨*tnc* := *tnc* + 1;
       tn := *tnc*;
       active := *active*—{*id of this transaction*}⟩;
      (*cleanup*))
    else (
      ⟨*active* := *active*—{*id of transaction*}⟩;
      (*backup*))).

TransactionMgr

```
public void finishedCommit(Set<RecordField> writeSet) {
    synchronized(tncLock) {
        tnc++;
        writeSets.put(tnc, new HashSet<RecordField>(writeSet));
    }
}
```

Transaction

```
public void commit() {

    if(VanillaDb.txMgr().validate(this)) {
        commitCertify();
        commitWorkspace();
        for (TransactionLifecycleListener l : lifecycleListeners)
            l.onTxCommit(this);

        if (logger.isLoggable(Level.FINE))
            logger.fine("transaction " + txNum + " committed");

        VanillaDb.txMgr().finishedCommit(writeSet.keySet());
        VanillaDb.txMgr().endCommit(this);
    } else {
        VanillaDb.txMgr().endCommit(this);
        throw new ValidAbortException("abort tx." + this.txNum + " for no validation");
    }

}
```

add to txMgr write sets

end validation

```
tend = (
   ⟨finish tn := tnc;
    finish active := (make a copy of active);
    active := active ∪ {id of this transaction}⟩;
    valid := true;
for t from start tn + 1 to finish tn do
      if (write set of transaction with transaction number t intersects read set)
        then valid := false;
for i ∈ finish active do
      if (write set of transaction Ti intersects read set or write set)
        then valid := false;
if valid
      then (
        (write phase);
        ⟨tnc := tnc + 1;
         tn := tnc;
         active := active—{id of this transaction}⟩;
        (cleanup))
      else (
        ⟨active := active—{id of transaction}⟩;
        (backup))).
```

TransactionMgr

```
public void endCommit(Transaction tx) {
    synchronized(activeLock) {
        activeSets.remove(tx);
    }
}
```

Transaction

```
public void commit() {

    if(VanillaDb.txMgr().validate(this)) {
        commitCertify();
        commitWorkspace();
        for (TransactionLifecycleListener l : lifecycleListeners)
            l.onTxCommit(this);

        if (logger.isLoggable(Level.FINE))
            logger.fine("transaction " + txNum + " committed");

        VanillaDb.txMgr().finishedCommit(writeSet.keySet());
        VanillaDb.txMgr().endCommit(this);
    } else {
        VanillaDb.txMgr().endCommit(this);
        throw new ValidAbortException("abort tx." + this.txNum + " for no validation");
    }

}
```

remove itself from active set

end validation

```
tend = (
   ⟨finish tn := tnc;
    finish active := (make a copy of active);
    active := active ∪ {id of this transaction}⟩;
    valid := true;
for t from start tn + 1 to finish tn do
      if (write set of transaction with transaction number t intersects read set)
        then valid := false;
for i ∈ finish active do
      if (write set of transaction Ti intersects read set or write set)
        then valid := false;
if valid
      then (
        (write phase);
        ⟨tnc := tnc + 1;
         tn := tnc;
         active := active—{id of this transaction}⟩;
        (cleanup))
      else (
        ⟨active := active—{id of transaction}⟩;
        ⟨backup⟩)).
```

Transaction

```
public void commit() {

    if(VanillaDb.txMgr().validate(this)) {
        commitCertify();
        commitWorkspace();
        for (TransactionLifecycleListener l : lifecycleListeners)
            l.onTxCommit(this);

        if (logger.isLoggable(Level.FINE))
            logger.fine("transaction " + txNum + " committed");

        VanillaDb.txMgr().finishedCommit(writeSet.keySet());
        VanillaDb.txMgr().endCommit(this);
    } else {
        VanillaDb.txMgr().endCommit(this);
        throw new ValidAbortException("abort tx." + this.txNum + " for no validation");
    }
}
```

no validation throw an error and will be rollback

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
public void commit() {
    if(VanillaDb.txMgr().validate(this)) {
        commitCertify();
        commitWorkspace();
        for (TransactionLifecycleListener l : lifecycleListeners)
            l.onTxCommit(this);

        if (logger.isLoggable(Level.FINE))
            logger.fine("transaction " + txNum + " committed");

        VanillaDb.txMgr().finishedCommit(writeSet.keySet());
        VanillaDb.txMgr().endCommit(this);
    } else {
        VanillaDb.txMgr().endCommit(this);
        throw new ValidAbortException("abort tx." + this.txNum + " for no validation");
    }

}
```

```
public void commitCertify() {
    this.cCertified = true;
}
```

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
public void commit() {
    if(VanillaDb.txMgr().validate(this)) {
        commitCertify();
        commitWorkspace();
        for (TransactionLifecycleListener l : lifecycleListeners)
            l.onTxCommit(this);

        if (logger.isLoggable(Level.FINE))
            logger.fine("transaction " + txNum + " committed");

        VanillaDb.txMgr().finishedCommit(writeSet.keySet());
        VanillaDb.txMgr().endCommit(this);
    } else {
        VanillaDb.txMgr().endCommit(this);
        throw new ValidAbortException("abort tx." + this.txNum + " for no validation");
    }

}
```

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: writeSet.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val = entry.getValue();
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.moveToRecordId(rfield.rid);
        rfile.setVal(rfield.fldName, val);
        rfile.close();
    }

    for(RecordField rfield: deleteSet) {
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.delete(rfield.rid);
        rfile.close();
    }
}
```

get a RecordFile of the table to be modified

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: writeSet.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val = entry.getValue();
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.moveToRecordId(rfield.rid);
        rfile.setVal(rfield.fldName, val);
        rfile.close();
    }

    for(RecordField rfield: deleteSet) {
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.delete(rfield.rid);
        rfile.close();
    }
}
```

modify through the original path

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: writeSet.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val = entry.getValue();
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.moveToRecordId(rfield.rid);
        rfile.setVal(rfield.fldName, val);
        rfile.close();
    }

    for(RecordField rfield: deleteSet) {
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.delete(rfield.rid);
        rfile.close();
    }
}
```

RecordFile

```
public void setVal(String fldName, Constant val) {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();
    Type fldType = ti.schema().type(fldName);

    Constant v = val.castTo(fldType);
    if (Page.size(v) > Page.maxSize(fldType))
        throw new SchemaIncompatibleException();
    if (!tx.rollbackCertified() && !tx.commitCertified() && !isTempTable()) {
        tx.putVal(ti.tableName(), currentRecordId(), fldName, v);
    } else {
        rp.setVal(fldName, v);
    }
}
```

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
private void commitWorkspace() {
    for (Map.Entry<RecordField, Constant> entry: writeSet.entrySet()) {
        RecordField rfield = entry.getKey();
        Constant val = entry.getValue();
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.moveToRecordId(rfield.rid);
        rfile.setVal(rfield.fldName, val);
        rfile.close();
    }

    for(RecordField rfield: deleteSet) {
        TableInfo ti = VanillaDb.catalogMgr().getTableInfo(rfield.tblName, this);
        RecordFile rfile = ti.open(this, true);
        rfile.delete(rfield.rid);
        rfile.close();
    }
}
```

modify through the original path

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
private void commitWorksp
    for (Map.Entry<Record
        RecordField rfiel
        Constant val = en
        TableInfo ti = Va
        RecordFile rfile
        rfile.moveToRecor
        rfile.setVal(rfie
        rfile.close();
    }

    for(RecordField rfiel
        TableInfo ti = Va
        RecordFile rfile
        rfile.delete(rfie
        rfile.close();
    }
}
```

RecordFile

```
public void delete() {
    if (tx.isReadOnly() && !isTempTable())
        throw new UnsupportedOperationException();

if(!tx.rollbackCertified() && !tx.commitCertified()) {
    tx.delete(ti.tableName(), currentRecordId());
} else {

    if (fhp == null)
        fhp = openHeaderForModification();

    // Log that this logical operation starts
    RecordId deletedRid = currentRecordId();
    tx.recoveryMgr().logLogicalStart();

    // Delete the current record
    rp.delete(fhp.getLastDeletedSlot());
    fhp.setLastDeletedSlot(currentRecordId());

    // Log that this logical operation ends
    tx.recoveryMgr().logRecordFileDeletionEnd(ti.tableName(), deletedRid.block().number(), deletedRid.id());

    // Close the header (release the header lock)
    closeHeader();

}
}
```

write phase

for $n \in$ *write set* **do** *exchange*$(n, copies[n])$.
for $n \in$ *delete set* **do** *delete*(n);

Transaction

```
public void rollback() {
    rollbackCertify();

    for (TransactionLifecycleListener l : lifecycleListeners) {

        l.onTxRollback(this);
    }

    if (logger.isLoggable(Level.FINE))
        logger.fine("transaction " + txNum + " rolled back");
}
```

```
public void rollbackCertify() {
    this.rCertified = true;
}
```

Outline

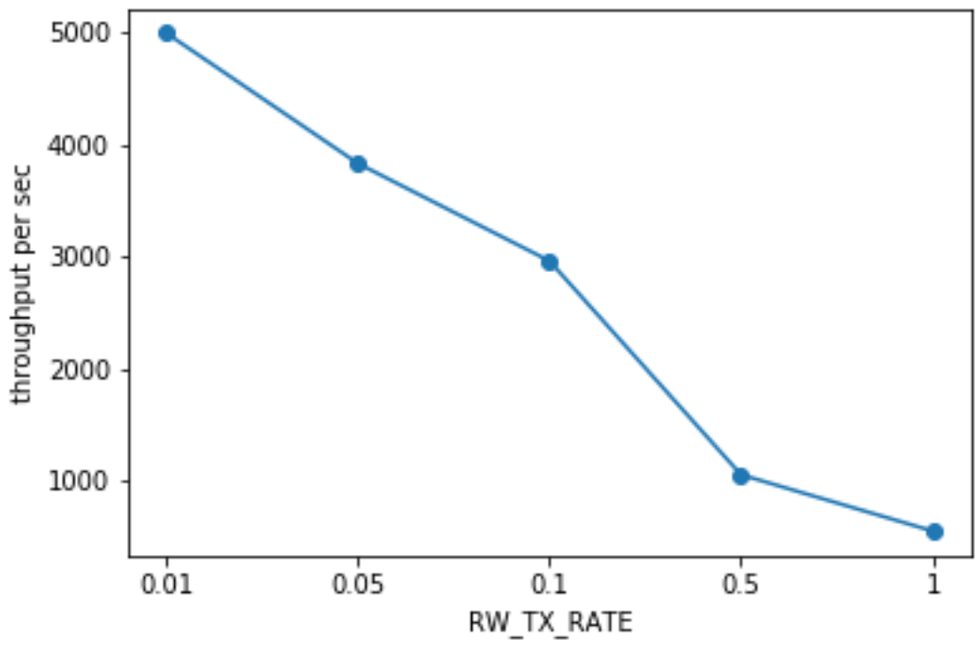
- Paper introduction
  - Motivation and Problem
  - Main Idea
  - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

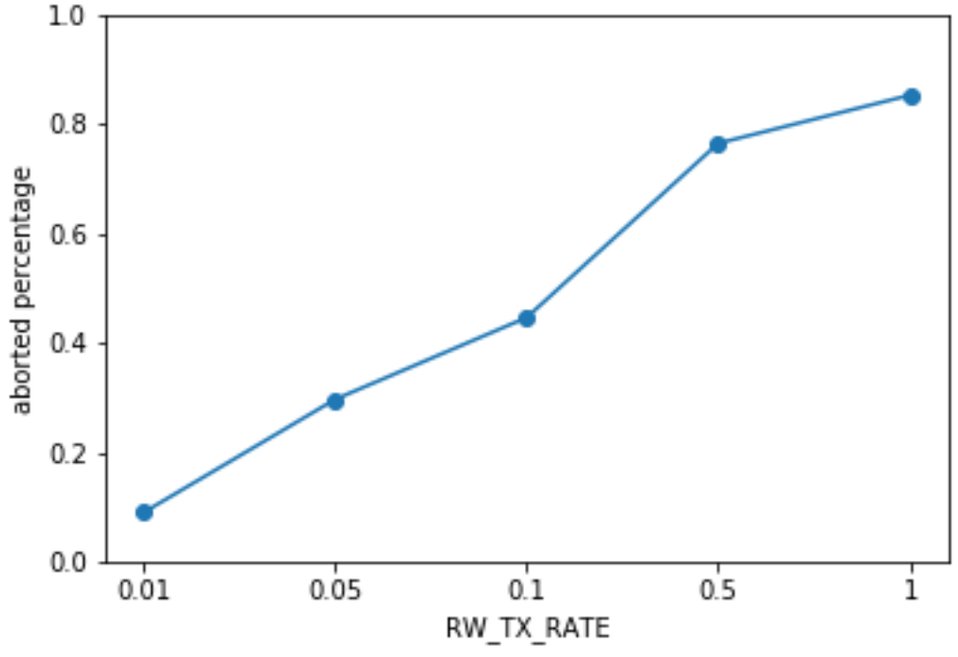
Evaluation and Experiment

- environment

Evaluation and Experiment

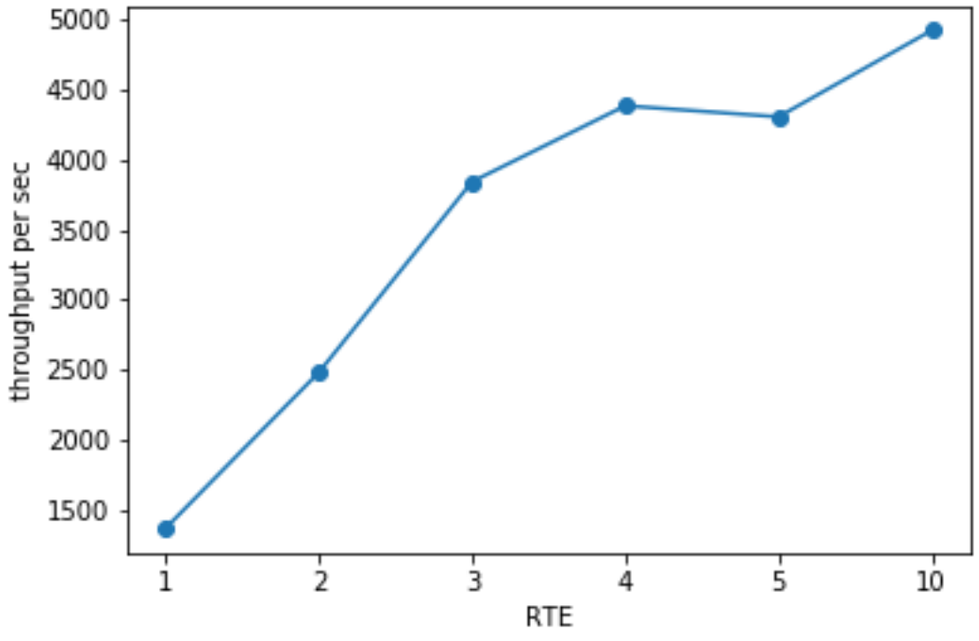
- Exp. I - Change RW_TX_RATE to verify

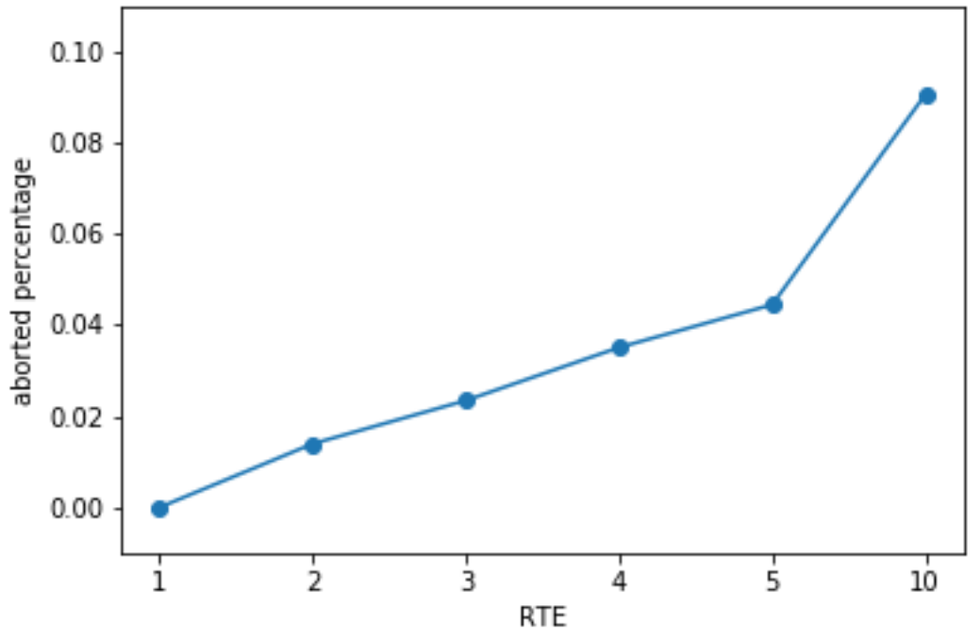




Evaluation and Experiment

- Exp. II - Change RTE to verify





Outline

- Paper introduction
  - Motivation and Problem
  - Main Idea
  - Conclusion
- Implementation
- Evaluation and Experiments
- Conclusion

Conclusion

- the paper did not mention the detail, so we spend more time on thinking what will happen.

- our OCC experiment shows that the curve is fit the OCC property.