MySQL-Doc

2022年8月24日　　15:42

==========

https://dev.mysql.com/doc/refman/8.0/en/

MySQL 8.0 Reference Manual

https://dev.mysql.com/doc/refman/8.0/en/features.html
c,c++ 写的。 cmake 配置编译的。

An EXPLAIN statement to show how the optimizer resolves a query.

每个表 最多支持 64个索引。每个索引可能由 1到16列 或 列的部分 组成。
InnoDB的 最大索引宽度 是 767 或 3072 字节。
MyISAM 最大index width 是 1000字节。
索引可以是 char,varchar,blob,text 类型的 列 的 前缀。

https://dev.mysql.com/doc/refman/8.0/en/mysql-nutshell.html
1.3 What Is New in MySQL 8.0

。。。太多了。。。

。。太多了。。。擦。。

pdf下下来，6000多页。。。

https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-innodb-cluster.html

---

https://dev.mysql.com/doc/refman/8.0/en/built-in-function-reference.html

Table 12.1 Built-In Functions and Operators

Name	Description	Introduced	Deprecated
&	Bitwise AND		
>	Greater than operator		
>>	Right shift		
>=	Greater than or equal operator		
<	Less than operator		
<>, !=	Not equal operator		

<< Left shift
<= Less than or equal operator
<=> NULL-safe equal to operator
。。 = 两侧出现 null时，返回null， <=> 两侧都是null时返回1，一侧null返回0.

%, MOD Modulo operator
* Multiplication operator
+ Addition operator
- Minus operator
- Change the sign of the argument
-> Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT().
->> Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()).
/ Division operator
:= Assign a value
= Assign a value (as part of a SET statement, or as part of the SET clause in an UPDATE statement)
= Equal operator
^ Bitwise XOR
ABS() Return the absolute value
ACOS() Return the arc cosine
ADDDATE() Add time values (intervals) to a date value
ADDTIME() Add time
AES_DECRYPT() Decrypt using AES
AES_ENCRYPT() Encrypt using AES
AND, && Logical AND
ANY_VALUE() Suppress ONLY_FULL_GROUP_BY value rejection
ASCII() Return numeric value of left-most character

ASIN()	Return the arc sine
ATAN()	Return the arc tangent
ATAN2(), ATAN()	Return the arc tangent of the two arguments
AVG()	Return the average value of the argument
BENCHMARK()	Repeatedly execute an expression
BETWEEN ... AND ...	Whether a value is within a range of values
BIN()	Return a string containing binary representation of a number

BIN_TO_UUID()	Convert binary UUID to string
BINARY	Cast a string to a binary string	8.0.27
BIT_AND()	Return bitwise AND
BIT_COUNT()	Return the number of bits that are set
BIT_LENGTH()	Return length of argument in bits
BIT_OR()	Return bitwise OR
BIT_XOR()	Return bitwise XOR
CAN_ACCESS_COLUMN()	Internal use only
CAN_ACCESS_DATABASE()	Internal use only
CAN_ACCESS_TABLE()	Internal use only
CAN_ACCESS_USER()	Internal use only	8.0.22
CAN_ACCESS_VIEW()	Internal use only
CASE	Case operator
CAST()	Cast a value as a certain type
CEIL()	Return the smallest integer value not less than the argument

CEILING()	Return the smallest integer value not less than the argument

CHAR()	Return the character for each integer passed
CHAR_LENGTH()	Return number of characters in argument
CHARACTER_LENGTH()	Synonym for CHAR_LENGTH()
CHARSET()	Return the character set of the argument
COALESCE()	Return the first non-NULL argument
COERCIBILITY()	Return the collation coercibility value of the string argument

COLLATION()	Return the collation of the string argument
COMPRESS()	Return result as a binary string
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenate with separator
CONNECTION_ID()	Return the connection ID (thread ID) for the connection

CONV()	Convert numbers between different number bases
CONVERT()	Cast a value as a certain type
CONVERT_TZ()	Convert from one time zone to another
COS()	Return the cosine
COT()	Return the cotangent
COUNT()	Return a count of the number of rows returned
COUNT(DISTINCT)	Return the count of a number of different values
CRC32()	Compute a cyclic redundancy check value
CUME_DIST()	Cumulative distribution value
CURDATE()	Return the current date

CURRENT_DATE(), CURRENT_DATE Synonyms for CURDATE()
CURRENT_ROLE()   Return the current active roles
CURRENT_TIME(), CURRENT_TIME Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP   Synonyms for NOW()
CURRENT_USER(), CURRENT_USER The authenticated user name and host name

CURTIME()   Return the current time
DATABASE() Return the default (current) database name
DATE()   Extract the date part of a date or datetime expression
DATE_ADD() Add time values (intervals) to a date value
DATE_FORMAT()   Format date as specified
DATE_SUB() Subtract a time value (interval) from a date
DATEDIFF() Subtract two dates
DAY()   Synonym for DAYOFMONTH()
DAYNAME()   Return the name of the weekday
DAYOFMONTH()   Return the day of the month (0-31)
DAYOFWEEK()   Return the weekday index of the argument
DAYOFYEAR()   Return the day of the year (1-366)
DEFAULT()   Return the default value for a table column
DEGREES()   Convert radians to degrees
DENSE_RANK()   Rank of current row within its partition, without gaps

DIV   Integer division
ELT()   Return string at index number
EXP()   Raise to the power of
EXPORT_SET()   Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
EXTRACT()   Extract part of a date
ExtractValue()   Extract a value from an XML string using XPath notation

FIELD()   Index (position) of first argument in subsequent arguments
FIND_IN_SET()   Index (position) of first argument within second argument

FIRST_VALUE()   Value of argument from first row of window frame
FLOOR()   Return the largest integer value not greater than the argument

FORMAT()   Return a number formatted to specified number of decimal places

FORMAT_BYTES()   Convert byte count to value with units   8.0.16
FORMAT_PICO_TIME()   Convert time in picoseconds to value with units
      8.0.16
FOUND_ROWS()   For a SELECT with a LIMIT clause, the number of rows that would be returned were there no LIMIT clause
FROM_BASE64()   Decode base64 encoded string and return result
FROM_DAYS()   Convert a day number to a date
FROM_UNIXTIME()   Format Unix timestamp as a date
GeomCollection() Construct geometry collection from geometries
GeometryCollection()   Construct geometry collection from geometries
GET_DD_COLUMN_PRIVILEGES()   Internal use only

GET_DD_CREATE_OPTIONS() Internal use only
GET_DD_INDEX_SUB_PART_LENGTH() Internal use only
GET_FORMAT() Return a date format string
GET_LOCK() Get a named lock
GREATEST() Return the largest argument
GROUP_CONCAT() Return a concatenated string
GROUPING() Distinguish super-aggregate ROLLUP rows from regular rows
GTID_SUBSET() Return true if all GTIDs in subset are also in set; otherwise false.
GTID_SUBTRACT() Return all GTIDs in set that are not in subset.
HEX() Hexadecimal representation of decimal or string value
HOUR() Extract the hour
ICU_VERSION() ICU library version
IF() If/else construct
IFNULL() Null if/else construct
IN() Whether a value is within a set of values
INET_ATON() Return the numeric value of an IP address
INET_NTOA() Return the IP address from a numeric value
INET6_ATON() Return the numeric value of an IPv6 address
INET6_NTOA() Return the IPv6 address from a numeric value
INSERT() Insert substring at specified position up to specified number of characters
INSTR() Return the index of the first occurrence of substring
INTERNAL_AUTO_INCREMENT() Internal use only
INTERNAL_AVG_ROW_LENGTH() Internal use only
INTERNAL_CHECK_TIME() Internal use only
INTERNAL_CHECKSUM() Internal use only
INTERNAL_DATA_FREE() Internal use only
INTERNAL_DATA_LENGTH() Internal use only
INTERNAL_DD_CHAR_LENGTH() Internal use only
INTERNAL_GET_COMMENT_OR_ERROR() Internal use only
INTERNAL_GET_ENABLED_ROLE_JSON() Internal use only 8.0.19
INTERNAL_GET_HOSTNAME() Internal use only 8.0.19
INTERNAL_GET_USERNAME() Internal use only 8.0.19
INTERNAL_GET_VIEW_WARNING_OR_ERROR() Internal use only
INTERNAL_INDEX_COLUMN_CARDINALITY() Internal use only
INTERNAL_INDEX_LENGTH() Internal use only
INTERNAL_IS_ENABLED_ROLE() Internal use only 8.0.19
INTERNAL_IS_MANDATORY_ROLE() Internal use only 8.0.19
INTERNAL_KEYS_DISABLED() Internal use only
INTERNAL_MAX_DATA_LENGTH() Internal use only
INTERNAL_TABLE_ROWS() Internal use only
INTERNAL_UPDATE_TIME() Internal use only
INTERVAL() Return the index of the argument that is less than the first argument

IS Test a value against a boolean
IS_FREE_LOCK() Whether the named lock is free
IS_IPV4() Whether argument is an IPv4 address
IS_IPV4_COMPAT() Whether argument is an IPv4-compatible address

IS_IPV4_MAPPED() Whether argument is an IPv4-mapped address
IS_IPV6() Whether argument is an IPv6 address
IS NOT Test a value against a boolean
IS NOT NULL NOT NULL value test
IS NULL NULL value test
IS_USED_LOCK() Whether the named lock is in use; return connection identifier if true
IS_UUID() Whether argument is a valid UUID
ISNULL() Test whether the argument is NULL
JSON_ARRAY() Create JSON array
JSON_ARRAY_APPEND() Append data to JSON document
JSON_ARRAY_INSERT() Insert into JSON array
JSON_ARRAYAGG() Return result set as a single JSON array
JSON_CONTAINS() Whether JSON document contains specific object at path

JSON_CONTAINS_PATH() Whether JSON document contains any data at path

JSON_DEPTH() Maximum depth of JSON document
JSON_EXTRACT() Return data from JSON document
JSON_INSERT() Insert data into JSON document
JSON_KEYS() Array of keys from JSON document
JSON_LENGTH() Number of elements in JSON document
JSON_MERGE() Merge JSON documents, preserving duplicate keys. Deprecated synonym for JSON_MERGE_PRESERVE() Yes
JSON_MERGE_PATCH() Merge JSON documents, replacing values of duplicate keys

JSON_MERGE_PRESERVE() Merge JSON documents, preserving duplicate keys

JSON_OBJECT() Create JSON object
JSON_OBJECTAGG() Return result set as a single JSON object
JSON_OVERLAPS() Compares two JSON documents, returns TRUE (1) if these have any key-value pairs or array elements in common, otherwise FALSE (0) 8.0.17
JSON_PRETTY() Print a JSON document in human-readable format
JSON_QUOTE() Quote JSON document
JSON_REMOVE() Remove data from JSON document
JSON_REPLACE() Replace values in JSON document
JSON_SCHEMA_VALID() Validate JSON document against JSON schema; returns TRUE/1 if document validates against schema, or FALSE/0 if it does not 8.0.17
JSON_SCHEMA_VALIDATION_REPORT() Validate JSON document against JSON schema; returns report in JSON format on outcome on validation including success or failure and reasons for failure 8.0.17
JSON_SEARCH() Path to value within JSON document
JSON_SET() Insert data into JSON document
JSON_STORAGE_FREE() Freed space within binary representation of JSON column value following partial update
JSON_STORAGE_SIZE() Space used for storage of binary representation of a JSON document
JSON_TABLE() Return data from a JSON expression as a relational table

JSON_TYPE() Type of JSON value
JSON_UNQUOTE() Unquote JSON value
JSON_VALID() Whether JSON value is valid
JSON_VALUE() Extract value from JSON document at location pointed to by path provided; return this value as VARCHAR(512) or specified type 8.0.21
LAG() Value of argument from row lagging current row within partition

LAST_DAY Return the last day of the month for the argument
LAST_INSERT_ID() Value of the AUTOINCREMENT column for the last INSERT

LAST_VALUE() Value of argument from last row of window frame
LCASE() Synonym for LOWER()
LEAD() Value of argument from row leading current row within partition

LEAST() Return the smallest argument
LEFT() Return the leftmost number of characters as specified
LENGTH() Return the length of a string in bytes
LIKE Simple pattern matching
LineString() Construct LineString from Point values
LN() Return the natural logarithm of the argument
LOAD_FILE() Load the named file
LOCALTIME(), LOCALTIME Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP() Synonym for NOW()
LOCATE() Return the position of the first occurrence of substring
LOG() Return the natural logarithm of the first argument
LOG10() Return the base-10 logarithm of the argument
LOG2() Return the base-2 logarithm of the argument
LOWER() Return the argument in lowercase
LPAD() Return the string argument, left-padded with the specified string

LTRIM() Remove leading spaces
MAKE_SET() Return a set of comma-separated strings that have the corresponding bit in bits set
MAKEDATE() Create a date from the year and day of year
MAKETIME() Create time from hour, minute, second
MASTER_POS_WAIT() Block until the replica has read and applied all updates up to the specified position 8.0.26
MATCH() Perform full-text search
MAX() Return the maximum value
MBRContains() Whether MBR of one geometry contains MBR of another
MBRCoveredBy() Whether one MBR is covered by another
MBRCovers() Whether one MBR covers another
MBRDisjoint() Whether MBRs of two geometries are disjoint
MBREquals() Whether MBRs of two geometries are equal
MBRIntersects() Whether MBRs of two geometries intersect
MBROverlaps() Whether MBRs of two geometries overlap
MBRTouches() Whether MBRs of two geometries touch
MBRWithin() Whether MBR of one geometry is within MBR of another
MD5() Calculate MD5 checksum

MEMBER OF() Returns true (1) if first operand matches any element of JSON array passed as second operand, otherwise returns false (0) 8.0.17
MICROSECOND() Return the microseconds from argument
MID() Return a substring starting from the specified position
MIN() Return the minimum value
MINUTE() Return the minute from the argument
MOD() Return the remainder
MONTH() Return the month from the date passed
MONTHNAME() Return the name of the month
MultiLineString() Contruct MultiLineString from LineString values

MultiPoint() Construct MultiPoint from Point values
MultiPolygon() Construct MultiPolygon from Polygon values
NAME_CONST() Cause the column to have the given name
NOT, ! Negates value
NOT BETWEEN ... AND ... Whether a value is not within a range of values

NOT IN() Whether a value is not within a set of values
NOT LIKE Negation of simple pattern matching
NOT REGEXP Negation of REGEXP
NOW() Return the current date and time
NTH_VALUE() Value of argument from N-th row of window frame
NTILE() Bucket number of current row within its partition.
NULLIF() Return NULL if expr1 = expr2
OCT() Return a string containing octal representation of a number

OCTET_LENGTH() Synonym for LENGTH()
OR, || Logical OR
ORD() Return character code for leftmost character of the argument

PERCENT_RANK() Percentage rank value
PERIOD_ADD() Add a period to a year-month
PERIOD_DIFF() Return the number of months between periods
PI() Return the value of pi
Point() Construct Point from coordinates
Polygon() Construct Polygon from LineString arguments
POSITION() Synonym for LOCATE()
POW() Return the argument raised to the specified power
POWER() Return the argument raised to the specified power
PS_CURRENT_THREAD_ID() Performance Schema thread ID for current thread 8.0.16
PS_THREAD_ID() Performance Schema thread ID for given thread 8.0.16
QUARTER() Return the quarter from a date argument
QUOTE() Escape the argument for use in an SQL statement
RADIANS() Return argument converted to radians
RAND() Return a random floating-point value
RANDOM_BYTES() Return a random byte vector
RANK() Rank of current row within its partition, with gaps
REGEXP Whether string matches regular expression

REGEXP_INSTR() Starting index of substring matching regular expression

REGEXP_LIKE() Whether string matches regular expression
REGEXP_REPLACE() Replace substrings matching regular expression
REGEXP_SUBSTR() Return substring matching regular expression
RELEASE_ALL_LOCKS() Release all current named locks
RELEASE_LOCK() Release the named lock
REPEAT() Repeat a string the specified number of times
REPLACE() Replace occurrences of a specified string
REVERSE() Reverse the characters in a string
RIGHT() Return the specified rightmost number of characters
RLIKE Whether string matches regular expression
ROLES_GRAPHML() Return a GraphML document representing memory role subgraphs

ROUND() Round the argument
ROW_COUNT() The number of rows updated
ROW_NUMBER() Number of current row within its partition
RPAD() Append string the specified number of times
RTRIM() Remove trailing spaces
SCHEMA() Synonym for DATABASE()
SEC_TO_TIME() Converts seconds to 'hh:mm:ss' format
SECOND() Return the second (0-59)
SESSION_USER() Synonym for USER()
SHA1(), SHA() Calculate an SHA-1 160-bit checksum
SHA2() Calculate an SHA-2 checksum
SIGN() Return the sign of the argument
SIN() Return the sine of the argument
SLEEP() Sleep for a number of seconds
SOUNDEX() Return a soundex string
SOUNDS LIKE Compare sounds
SOURCE_POS_WAIT() Block until the replica has read and applied all updates up to the specified position 8.0.26
SPACE() Return a string of the specified number of spaces
SQRT() Return the square root of the argument
ST_Area() Return Polygon or MultiPolygon area
ST_AsBinary(), ST_AsWKB() Convert from internal geometry format to WKB

ST_AsGeoJSON() Generate GeoJSON object from geometry
ST_AsText(), ST_AsWKT() Convert from internal geometry format to WKT

ST_Buffer() Return geometry of points within given distance from geometry

ST_Buffer_Strategy() Produce strategy option for ST_Buffer()
ST_Centroid() Return centroid as a point
ST_Collect() Aggregate spatial values into collection 8.0.24
ST_Contains() Whether one geometry contains another
ST_ConvexHull() Return convex hull of geometry
ST_Crosses() Whether one geometry crosses another
ST_Difference() Return point set difference of two geometries

ST_Dimension() Dimension of geometry
ST_Disjoint() Whether one geometry is disjoint from another
ST_Distance() The distance of one geometry from another
ST_Distance_Sphere() Minimum distance on earth between two geometries

ST_EndPoint() End Point of LineString
ST_Envelope() Return MBR of geometry
ST_Equals() Whether one geometry is equal to another
ST_ExteriorRing() Return exterior ring of Polygon
ST_FrechetDistance() The discrete Fréchet distance of one geometry from another 8.0.23
ST_GeoHash() Produce a geohash value
ST_GeomCollFromText(), ST_GeometryCollectionFromText(), ST_GeomCollFromTxt() Return geometry collection from WKT
ST_GeomCollFromWKB(), ST_GeometryCollectionFromWKB() Return geometry collection from WKB
ST_GeometryN() Return N-th geometry from geometry collection
ST_GeometryType() Return name of geometry type
ST_GeomFromGeoJSON() Generate geometry from GeoJSON object
ST_GeomFromText(), ST_GeometryFromText() Return geometry from WKT
ST_GeomFromWKB(), ST_GeometryFromWKB() Return geometry from WKB
ST_HausdorffDistance() The discrete Hausdorff distance of one geometry from another 8.0.23
ST_InteriorRingN() Return N-th interior ring of Polygon
ST_Intersection() Return point set intersection of two geometries

ST_Intersects() Whether one geometry intersects another
ST_IsClosed() Whether a geometry is closed and simple
ST_IsEmpty() Whether a geometry is empty
ST_IsSimple() Whether a geometry is simple
ST_IsValid() Whether a geometry is valid
ST_LatFromGeoHash() Return latitude from geohash value
ST_Latitude() Return latitude of Point 8.0.12
ST_Length() Return length of LineString
ST_LineFromText(), ST_LineStringFromText() Construct LineString from WKT

ST_LineFromWKB(), ST_LineStringFromWKB() Construct LineString from WKB

ST_LineInterpolatePoint() The point a given percentage along a LineString 8.0.24
ST_LineInterpolatePoints() The points a given percentage along a LineString 8.0.24
ST_LongFromGeoHash() Return longitude from geohash value
ST_Longitude() Return longitude of Point 8.0.12
ST_MakeEnvelope() Rectangle around two points
ST_MLineFromText(), ST_MultiLineStringFromText() Construct MultiLineString from WKT
ST_MLineFromWKB(), ST_MultiLineStringFromWKB() Construct MultiLineString from WKB

ST_MPointFromText(), ST_MultiPointFromText() Construct MultiPoint from WKT

ST_MPointFromWKB(), ST_MultiPointFromWKB() Construct MultiPoint from WKB

ST_MPolyFromText(), ST_MultiPolygonFromText() Construct MultiPolygon from WKT

ST_MPolyFromWKB(), ST_MultiPolygonFromWKB() Construct MultiPolygon from WKB

ST_NumGeometries() Return number of geometries in geometry collection

ST_NumInteriorRing(), ST_NumInteriorRings() Return number of interior rings in Polygon
ST_NumPoints() Return number of points in LineString
ST_Overlaps() Whether one geometry overlaps another
ST_PointAtDistance() The point a given distance along a LineString 8.0.24

ST_PointFromGeoHash() Convert geohash value to POINT value
ST_PointFromText() Construct Point from WKT
ST_PointFromWKB() Construct Point from WKB
ST_PointN() Return N-th point from LineString
ST_PolyFromText(), ST_PolygonFromText() Construct Polygon from WKT
ST_PolyFromWKB(), ST_PolygonFromWKB() Construct Polygon from WKB
ST_Simplify() Return simplified geometry
ST_SRID() Return spatial reference system ID for geometry
ST_StartPoint() Start Point of LineString
ST_SwapXY() Return argument with X/Y coordinates swapped
ST_SymDifference() Return point set symmetric difference of two geometries

ST_Touches() Whether one geometry touches another
ST_Transform() Transform coordinates of geometry 8.0.13
ST_Union() Return point set union of two geometries
ST_Validate() Return validated geometry
ST_Within() Whether one geometry is within another
ST_X() Return X coordinate of Point
ST_Y() Return Y coordinate of Point
STATEMENT_DIGEST() Compute statement digest hash value
STATEMENT_DIGEST_TEXT() Compute normalized statement digest
STD() Return the population standard deviation
STDDEV() Return the population standard deviation
STDDEV_POP() Return the population standard deviation
STDDEV_SAMP() Return the sample standard deviation
STR_TO_DATE() Convert a string to a date
STRCMP() Compare two strings
SUBDATE() Synonym for DATE_SUB() when invoked with three arguments
SUBSTR() Return the substring as specified
SUBSTRING() Return the substring as specified
SUBSTRING_INDEX() Return a substring from a string before the specified number of occurrences of the delimiter
SUBTIME() Subtract times

SUM()	Return the sum
SYSDATE()	Return the time at which the function executes
SYSTEM_USER()	Synonym for USER()
TAN()	Return the tangent of the argument
TIME()	Extract the time portion of the expression passed
TIME_FORMAT()	Format as time
TIME_TO_SEC()	Return the argument converted to seconds
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPADD()	Add an interval to a datetime expression
TIMESTAMPDIFF()	Subtract an interval from a datetime expression
TO_BASE64()	Return the argument converted to a base-64 string
TO_DAYS()	Return the date argument converted to days
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0
TRIM()	Remove leading and trailing spaces
TRUNCATE()	Truncate to specified number of decimal places
UCASE()	Synonym for UPPER()
UNCOMPRESS()	Uncompress a string compressed
UNCOMPRESSED_LENGTH()	Return the length of a string before compression

UNHEX()	Return a string containing hex representation of a number
UNIX_TIMESTAMP()	Return a Unix timestamp
UpdateXML()	Return replaced XML fragment
UPPER()	Convert to uppercase
USER()	The user name and host name provided by the client
UTC_DATE()	Return the current UTC date
UTC_TIME()	Return the current UTC time
UTC_TIMESTAMP()	Return the current UTC date and time
UUID()	Return a Universal Unique Identifier (UUID)
UUID_SHORT()	Return an integer-valued universal identifier
UUID_TO_BIN()	Convert string UUID to binary
VALIDATE_PASSWORD_STRENGTH()	Determine strength of password
VALUES()	Define the values to be used during an INSERT
VAR_POP()	Return the population standard variance
VAR_SAMP()	Return the sample variance
VARIANCE()	Return the population standard variance
VERSION()	Return a string that indicates the MySQL server version
WAIT_FOR_EXECUTED_GTID_SET()	Wait until the given GTIDs have executed on the replica.
WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS()	Use WAIT_FOR_EXECUTED_GTID_SET(). 8.0.18
WEEK()	Return the week number
WEEKDAY()	Return the weekday index
WEEKOFYEAR()	Return the calendar week of the date (1-53)
WEIGHT_STRING()	Return the weight string for a string
XOR	Logical XOR
YEAR()	Return the year

YEARWEEK() Return the year and week
| Bitwise OR
~   Bitwise inversion

==========

==========

https://dev.mysql.com/doc/refman/8.0/en/optimization.html

优化

性能在 数据库层面 依赖于多个因素，如 表，查询，配置。 这些最终作用于 CPU 和 IO，你希望这些 越低越好。

在处理数据库性能时，首先要学习 软件方面的 高级规则和指南， 使用 耗时 来衡量性能。
当你称为专家后，你会更多地 了解内部发生的事情，并开始测量 CPU IO 之类的东西。

典型用户的目标 是从 现有软件和硬件中 获得 最佳数据库性能。
高级用户 寻找机会 改进MySQL 软件本身，或开发自己的 存储引擎 和硬件设备 以扩展MySQL 生态系统。

数据库层 优化
使数据库快的 最重要因素 是 基本设计：

1. 表结构是否正确？特别是，列是否具有正确的数据类型，每个表是否具有 适合工作类型的列？例如，update频繁的 应用 通常有很多表 和 很少的列。 分析大量数据的 应用 通常由很少的表 和 很多的列
2. 是否有正确的索引 来提高 查询效率？

3. 是否为每个表 使用了 适当的 存储引擎，并利用了 你使用的每个存储引擎的 优势和特性？特别是，选择 事务存储引擎(InnoDB) 和 非事务存储引擎(MyISAM) 对于 性能和扩展非常重要。
4. 每个表是否使用 适当的 行格式(row format)？ 此选择还 依赖于 表的存储引擎。特别是，压缩后的表 使用更少的 磁盘空间，只需要更少的磁盘IO 来读取 和写入数据。压缩适用于InnoDB的所有类型的工作负载，以及 只读的 MyISAM表。
5. 应用是有使用了 适当的 locking strategy？例如，尽可能允许 共享访问，以便数据库操作 可以并发运行，并在适当时 请求独占访问，以便关键操作获得 最高优先级。 同样，存储引擎的选择很重要。InnoDB可以在你 不参与的情况下 处理大多数 锁定问题，从而在数据库中 实现更好的并发性，并减少 代码的试验 和 调整量。
6. 用于缓存的 内存区域的 大小是否正确？ 也就是说， 大到足以容纳 经常访问的数据，但又不会 大到 使 物理内存过载 并导致 分页。要配置的主要 内存区域是 InnoDB 缓冲池和 MyISAM 密钥缓存。

。。每个表 使用 适当的引擎。。
。。MySQL的缓存命中时 和 redis 比 速度怎么样？

硬件 优化
系统瓶颈通常来自：

1. 磁盘寻道。磁盘找到一条数据需要时间。对于现代磁盘，平均时间通常低于10ms，因此，理论上，我们每秒可以100次寻道。 使用新磁盘 会快一点点，很难针对 一个表进行优化。优化寻道时间的方法 是将 数据分布到 多个磁盘上。.
2. 磁盘读写。当磁盘在正确的位置时，我们需要读取或写入数据。使用现代磁盘，一个磁盘可以提供 至少 10-20MB/s 的吞吐量。这比查找更容易优化，因为你可以从多个磁盘并行读取。
3. CPU周期。当数据在主存中时，我们必须 对其进行 处理以获得 我们的结果。大表(相对于内存)是常见的制约因素。小表，速度通常不是问题。
4. 内存带宽。当CPU需要的数据 超出 CPU cache 的容量时， 内存带宽就成为了 瓶颈。对于大多数系统来说，这是一个 不常见的瓶颈，但需要注意。

。。SSD？ 不过，SSD的寿命会不会 很短。。数据库的 读写 比 一般个人电脑 的读写更猛吧。
。。多个磁盘。怎么搞的？

平衡 可移植性 和 性能
要在可移植的 MySQL 程序中 使用 面向 性能的 SQL扩展， 你可以将 MySQL 特定的关键字 包装在 /*! */ 中。 其他SQL服务器 忽略 注释的关键字。

优化SQL
数据库的核心逻辑是 通过SQL 执行的。
这里包含了 读取和写入数据的 SQL操作，一般SQL操作的幕后开销， 特定场景(如数据库监控)中使用的操作。

优化Select
Select语句的 查询的 优化 是最高优先级的。 无论是 亚秒级别的页面，还是 花数小时的 报

告。
除了select，查询的调优技术 也适用于 create table … as select, insert into … select, delete 语句中的where子句 等结构。 这些语句 有额外的性能考虑，因为它们结合了 写操作 和 面向读的 查询操作。

NDB cluster 支持 join pushdown (连接下推) 优化， 合格的join 会被 完整 发送到 NDB cluster 数据节点，在那里 可以被分布到它们之间 并发执行。

查询优化主要考虑：

1. 要使 select … where 更快， 首先要检查 是否可以添加索引。在where子句中使用到的列上加上索引，来加快 eval，filter，和最终 检索出 结果的 速度。 为了避免磁盘空间的浪费， construct a small set of indexes that speed up many related queries used in your application.
    a. 对于使用join 和 外键 来 连接不同表的查询，索引 尤其重要，你可以使用 explain 来确定 哪些索引 用于 select。
2. 隔离 和调整 查询的任何部分，例如 函数调用，这会花费过多的时间。根据查询的结构，一个函数 可能 对结果集的每行 调用一次， 也可能对 表中的每行 调用一次， 这极大的放大了 任何低效率。
3. 尽量减少查询中 全表扫描次数，尤其对于 大表。
4. 通过定期使用analyze table 使得 表统计信息 保持最新，使得 优化器 拥有 构建 有效执行计划 所需的 信息。
5. 了解 每个表的 存储引擎的 调优技术，索引技术 和 配置参数。 InnoDB 和 MyISAM 都有一套 指南 来 启用 和 维持 查询的 高性能。
6. 可以使用 InnoDB read-only transaction 来优化 InnoDB 表的 只读查询的事务。
7. 避免 以难以理解的方式转换查询，尤其是 在优化器自动执行某些相同的转换时。
8. 如果性能问题 不能通过 基本准则之一 轻松解决，请通过 阅读 explain 计划 并调整 索引，where子句，join子句 等 来调整 特定查询的 内部细节。
9. 调整MySQL 用于缓存的 内存区域的 大小 和属性。通过有效使用 InnoDB缓冲池，MyISAM key cache 和 MySQL 查询 cache，重复查询 运行地更快。
10. 即使对于 使用了缓存内存区域快速运行的查询，你依然可以进一步优化，以便它们需要更少的缓存内存，从而使你的应用 更具可扩展性。可扩展性意味着 你的应用可以处理 更多并发用户，更大的请求等，而不会出现 性能大幅度下降。
11. 处理 locking issue，你的查询速度 可能受到 同时访问表 的其他会话的 影响。

---

https://dev.mysql.com/doc/refman/8.0/en/where-optimization.html

where clause 优化

本节讨论 where子句的 优化， 例子是使用 select的， 但是 相同的 优化也可以应用于 delete 和 update 的 where子句中。

你可能想重写 查询 以使得 算术运算更快，同时牺牲 可读性。因为 MySQL 会自动进行类似的优化，所以 你通常可以避免这项工作，并将 查询保留在 更易于理解和维护的形式中。

MySQL执行的一些优化如下：

1. 移除不必要的括号

```
((a AND b) AND c OR (((a AND b) AND (c AND d))))
-> (a AND b AND c) OR (a AND b AND c AND d)
```

2. constant folding， 恒定折叠：

```
(a<b AND b=c) AND a=5
```

-> b>5 AND b=c AND a=5

3. constant condition removal, 恒定条件去除
   (b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)
   -> b=5 OR b=6
   在MySQL 8.0.14 及更高版本中， 这发生在 准备阶段 而不是 优化阶段，有助于简化 join
4. 索引使用的 常量表达式 只计算一次
5. 从8.0.16开始，对数值类型的列 和 常量值的比较 进行检查并折叠 或删除 无效 或超出范围的值。
   # CREATE TABLE t (c TINYINT UNSIGNED NOT NULL);
   SELECT * FROM t WHERE c << 256;
   ->> SELECT * FROM t WHERE 1;
6. count(*) 对单独一张表使用，并且没有 where 子句，MyISAM 和 MEMORY 会直接 从 表信息中 检索信息。 当只对一张表使用时，这也适用于任何 not null。
7. 早期检测无效的 常量表达式。MySQL 快速检测到 某些 select 语句 是不可能的，并且不会返回任何行。
8. 如果你不使用 group by 或 聚合函数 (count(),min()等)， having 将和 where 合并。
9. 对于 join 的每个表，构造一个 更简单的 where 以获得 对表的 快速 where 的eval，并尽快跳过行。
10. 在查询中，优先读取 所有常量表，然后其他表。 常量表是以下：
    a. 空表 或 只有一行的表
    b. 与 在 主键或 unique索引上 的where子句 一起使用的表，其中 所有 索引部分 都与常量表达式 进行比较 并定义为 not null . （A table that is used with a WHERE clause on a PRIMARY KEY or a UNIQUE index, where all index parts are compared to constant expressions and are defined as NOT NULL. ）
    c. 下面的表都是常量表
       SELECT * FROM t WHERE primary_key=1;
       SELECT * FROM t1,t2
         WHERE t1.primary_key=1 AND t2.primary_key=t1.id;
11. 通过尝试 所有可能性 来找到 用于 连接表的 最佳join组合。如果 order by 和 group by 子句中的 所有列都来自同一个表，则在 join 时 首选 该表。
12. 如果有一个 order by子句，和一个不同的 group by 子句，或者 如果order by 或 group by 包含 来自 join队列中 第一个表 以外的 表的 列，则会 创建一个 临时表。
13. 如果你使用了 SQL_SMALL_RESULT 修饰符，MySQL 使用内存表中的 临时表。
14. 查询每个 表索引，并使用 最佳索引，除非优化器 认为 使用 表扫描 更有效。曾经，根据最佳索引 是否 跨越超过30%的表 来使用扫描，但固定百分比 不再 决定 使用索引 还是 扫描。优化器 现在更加复杂，它的估计基于其他因素，例如，表大小，行数，IO块大小。
15. 有时，MySQL 甚至可以在 不查询 数据文件的情况下 从索引中读取行。如果 索引中 使用的 列都是数字，则仅使用索引树来解析查询。
16. 在输出每一行之前，会跳过那些与having子句不匹配的行。

下面的查询是非常快的

```
SELECT COUNT(*) FROM tbl_name;

SELECT MIN(key_part1),MAX(key_part1) FROM tbl_name;

SELECT MAX(key_part2) FROM tbl_name
  WHERE key_part1=constant;
```

```
SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... LIMIT 10;

SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... LIMIT 10;
```

MySQL 仅使用 索引树 解析下面的查询，假设 索引列 是数字

```
SELECT key_part1,key_part2 FROM tbl_name WHERE key_part1=val;

SELECT COUNT(*) FROM tbl_name
  WHERE key_part1=val1 AND key_part2=val2;

SELECT MAX(key_part2) FROM tbl_name GROUP BY key_part1;
```

下面的查询 使用 索引 来 有序地检索 每行，而不需要 单独的 sort步骤：

```
SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... ;

SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... ;
```

---

https://dev.mysql.com/doc/refman/8.0/en/range-optimization.html

range 优化

范围访问方法使用 单个 索引 来 检索 包含在 一个或多个 索引值 间隔内的 表行的 子集。它可用于 single-part 或 multiple-part 索引。

下面描述了 优化器使用 范围访问的 条件：

。。。什么是 single-part (整体的)，multiple-part (由几部分组成的) index
。。single-part 是指 范围访问 只需要 一个索引就可以 成功确认范围？

single-part 索引的 范围访问方法 (range access method for single-part indexes)
对于一个 single-part index，索引值区间 能 方便地用 where子句中 相应condition 表示。表示为 范围condition 而不是 interval。
single-part index 的 range condition 的定义如下：

1. 对于btree 和 hash 索引，在使用=, <=>, in(), is null, is not null 运算符时，将关键字部分 和 常量值 比较 是 range condition
2. 对于btree 索引，在使用 >, <, >=, <=, between, != , <> 时，将 key part 和 常量值 比较 是 range condition，或者 like 不以 通配符开头的 常量字符串。
3. 对于所有索引类型， 多个 range condition 可以通过 or , and 组成一个 range condition。

上面说到的 常量 是下面之一：

1. 来自查询字符串的 常量
2. 来自 同一个 join 的 const 或 system 表的 列
3. 不相关 子查询 的结果
4. 任何 完全由 上述类型的 子表达式 组成的 表达式

下面是 where子句中 使用了range condition 的例子

```
SELECT * FROM t1
  WHERE key_col > 1
  AND key_col < 10;

SELECT * FROM t1
  WHERE key_col = 1
  OR key_col IN (15,18,20);

SELECT * FROM t1
  WHERE key_col LIKE 'ab%'
  OR key_col BETWEEN 'bar' AND 'foo';
```

在优化器 constant propagation 阶段，一些 非常量值 可能会转换为 常量。

MySQL尝试从where子句中 为每个 可能的索引 提取 range condition。在提取过程中，丢弃不能用于 构建 范围条件 的 条件，合并产生 重叠方位的 条件，并去除 产生 空范围的 条件。

考虑以下语句，其中 key1 是索引列，nonkey 不是索引

```
SELECT * FROM t1 WHERE
  (key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
  (key1 < 'bar' AND nonkey = 4) OR
  (key1 < 'uux' AND key1 > 'z');
```

key1 的抽取过程如下：
从原始 where子句开始，

```
(key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
(key1 < 'bar' AND nonkey = 4) OR
(key1 < 'uux' AND key1 > 'z')
```

移除 nonkey=4 和 key1 like '%b' ，因为它们 不能用来 range scan，正确的方式是 移除它们，替换为 true，这样 在range scan 时 我们不会 miss 匹配的行。变成：

```
(key1 < 'abc' AND (key1 LIKE 'abcde%' OR TRUE)) OR
(key1 < 'bar' AND TRUE) OR
(key1 < 'uux' AND key1 > 'z')
```

折叠 始终true 或 false 的条件：

```
        (key1 LIKE 'abcde%' OR TRUE) is always true
        (key1 < 'uux' AND key1 > 'z') is always false
```

用true/false 替换 它们

(key1 < 'abc' AND TRUE) OR (key1 < 'bar' AND TRUE) OR (FALSE)

移除不必要的 true/false

(key1 < 'abc') OR (key1 < 'bar')

重叠的区间 合并成一个 用于 range scan 的 最终condition

(key1 < 'bar')

一般来说，range scan 的条件 不如 where子句 的严格，所以 MySQL 会执行额外的检查 来 filter 那么 符合 range scan 但不符合 where子句的 行。

range condition 提取算法 可以 处理 任意深度 的 嵌套的 and,or， 并且 其输出不依赖于 condition 在 where子句中的顺序。

MySQL 不支持 合并多个 空间索引(spatial indexes)的 range access 的 range，要解决这个限制，你可以使用 具有 相同 select 语句的 union，但你将 每个 spatial indexes 放在不同的 select中。

Range Access Method for Multiple-part Indexes

multiple-part index 的 range condition 是 single-part index 的 range condition 的扩展。多部分 索引 上的 range condition 将 索引行 限制在 一个或 多个 key tuple interval 内。key tuple interval 是 在一组 key tuple 上定义的，使用 index 中的 顺序。

例如，考虑一个 multiple-part index 定义为 (key_part1, key_part2, key_part3),下面是以key 为顺序的 key tuple。

key_part1	key_part2	key_part3
NULL	1	'abc'
NULL	1	'xyz'
NULL	2	'foo'
1	1	'abc'
1	1	'xyz'
1	2	'abc'
2	1	'aaa'

。。感觉是 联合索引啊。 不是。

条件 key_part1 = 1 定义了 下面的 interval：

(1,-inf,-inf) <= (key_part1,key_part2,key_part3) < (1,+inf,+inf)

这个interval 覆盖了 第4,5,6, 个tuple， 且能被用来 range access

条件 key_part3='abc' 没有定义 单个区间，并且 不能 被 range access method 使用。

下面更详细说明 range condition 如何 适用于 multi-part index：

对于hash 索引，可以使用包含 相同值 的 每个interval。这意味着 只能 针对 以下形式的 条件 生成 区间。

key_part1 cmp const1

AND key_part2 cmp const2
AND ...
AND key_partN cmp constN;

这里的 const1,const2等 是 常量， cmp 是 =, <=>, is null 之一， 且 这些condition 覆盖了 所有的 index part。 (这就是说，有N个 condition， 和 N-part index 的 每个 part 对应)。 例如，下面是一个 three-part hash index 的 range condition 。

key_part1 = 1 AND key_part2 IS NULL AND key_part3 = 'foo'

对于btree索引，interval 可能 可用于 与 and组合的 condition，其中每个 condition 使用 =,<=>,is null,>,<,>=,<=,!=,<>,between,like(不以通配符开头)。 一个interval 可用 只要 它可以 确定 包含 匹配条件的 所有行的 单个 key tuple。(如果使用<> 或 !=, 则可以使用 2个 间隔)

只要比较运算符是 =,<=>,is null， 优化器就会 尝试使用 其他 key part 来确认 interval。如果运算符是 >,<,>=,<=,!=,<>,between,like， 优化器 会使用它 但不再考虑 其他key part。对于下面的 表达式，优化器使用第一次比较中的 =。它还使用 第二个比较中的 >= 但不考虑其他 key-part，并且不使用 第三个 比较 进行 区间构造：

key_part1 = 'foo' AND key_part2 >= 10 AND key_part3 > 10

。。应该就是 联合索引，反正就是 一个索引 有 多个列。

single interval就是：

('foo',10,-inf) < (key_part1,key_part2,key_part3) < ('foo',+inf,+inf)

。。而且 就是 b-tree 搜索。

如果 覆盖了 区间中 行集合的 condition 被通过 or 组合在一起，它们会形成一个 condition，涵盖 包含在 它们的interval 并集中的 行集。 如果被 and 组合在一起，则它们会形成一个 condition，涵盖了 包含在它们的interval 交集中的 行集。

例如，对于 two-part index：

(key_part1 = 1 AND key_part2 < 2) OR (key_part1 > 5)

interval是：

(1,-inf) < (key_part1,key_part2) < (1,2)
(5,-inf) < (key_part1,key_part2)

在这个例子中，第一行的 interval 使用了 一个key part 组成 左边界， 使用 2个 key-part 组成 右界。 第二行的 interval，只使用了 一个 key-part。 explain 输出中的 key_len 列 指示 使用的 键前缀的 最大长度。

有些情况下，key_len 可能表示 使用的 key part,但这可能不是你所期望的。 假设 key_part1 和 key_part2 可以为 null， 然后 key_len 列 展示 以下condition 的 2个 key-part 的长度：

key_part1 >= 1 AND key_part2 < 2

但是，实际上，condition 被转换成下面：

key_part1 >= 1 AND key_part2 IS NOT NULL

Equality Range Optimization of Many-Valued Comparisons

考虑下面的表达式，其中 col_name 是一个 索引列：

col_name IN(val1, ..., valN)
col_name = val1 OR ... OR col_name = valN

如果 col_name 等于某个valX，每个表达式都是 true。 这些比较 等同于 range 比较(这里 range 是一个 单值)。 优化器通过 下面的方式 来估计 符合 等价的range比较 的 行 的成本：

如果 col_name 上存在 唯一索引，则每个 range 的行 估计值为1， 因为 最多一个行可以有 给定值。
否则，col_name 上的 任何索引都不是 唯一索引，优化器 可以dive into(探查) 索引 或索引统计信息 来估计 每个范围的 行数。

通过探查索引，优化器 在 range 的每一端 进行 探查，并使用 range 内的行数 作为 估计值。 例如，表达式 col_name in (10,20,30) ， 有3个 equality range，优化器 对每个 range 进行 2个 探查，来生成 行估计。 每对探查 都会 产生 具有给定值的 行数的 估计值。

index dive 提供了准确的 行估计，但随着 表达式中 比较值的数量 的增加， 优化器 需要 更长的时间 来生成 估计行。索引统计 不如 index dive 准确，但是 允许 对 large value list 进行更快的 行估计。

eq_range_index_dive_limit 系统变量 使你可以 配置 优化器 从 一种行估计策略 切换到 另一种的 界线值。要允许 对 最多 N个 equality range 来进行 index dive，那么设置 eq_range_index_dive_limit 为 N+1， 要禁用 索引统计，始终使用 index dive，那么 设置为 0。

使用 analyze table 来更新 table index statistics for 最好的估计。

MySQL 8.0之前，除了使用 eq_range_index_dive_limit之外， 没有办法 不使用index dive 来 估计 index usefulness。

MySQL 8.0中，满足下面的所有条件 的查询 可以跳过 index dive:

- 对 单表的查询， 而不是 多表的join
- 存在 single-index FORCE INDEX 索引提示。这个idea是，如果强制使用索引，那么执行 index dive 没有任何好处。
- 索引 是 非唯一的， 且不是 fulltext index
- 没有子查询
- 没有 distinct,group by, order by 子句。

对于 explain for connection， 如果 跳过 index dive ，输出会如下更改：

- 对于 traditional output (传统输出)，rows 和 filtered 值是 null
- 对于 json 输出，rows_examined_per_scan 和 rows_produced_per_join 没有出现，skip_index_dive_duo_to_force 为true， 成本计算不准确。

只有explain，不带for connection，输出不会有变化，即使 index dive 跳过。

在 执行了 跳过index dive 的 查询后，information_schema.optimizer_trace 表 中 相应的 行 包含 skipped_due_to_force_index 的 index_dives_for_range_access 值。

Skip Scan Range Access Method
考虑下面的场景

```
CREATE TABLE t1 (f1 INT NOT NULL, f2 INT NOT NULL, PRIMARY KEY(f1, f2));
INSERT INTO t1 VALUES
  (1,1), (1,2), (1,3), (1,4), (1,5),
  (2,1), (2,2), (2,3), (2,4), (2,5);
INSERT INTO t1 SELECT f1, f2 + 5 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 10 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 20 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 40 FROM t1;
ANALYZE TABLE t1;

EXPLAIN SELECT f1, f2 FROM t1 WHERE f2 > 40;
```

为了执行这个查询，MySQL 可以选择 index scan(索引扫描) 来 fetch 所有行(索引包含 所有要选择的列)，然后应用 where子句中的 f2>40 条件来 生成 最终的结果集。

range scan 比 full index scan 更高效，但 不能在 这种情况下 使用，因为 第一个index column f1 上没有 condition。
但是，从MySQL 8.0.13 开始，优化器可以执行 多个 range scan，每个 f1 值 一次 range scan. 使用了 被称为 skip scan 的方法， 类似于 loose index scan。

skip scan 适用于以下 condition：

1. 表 T 至少有一个 复合索引(compound index)，key part 的格式：
   ([A1,A2..AK],B1,B2..BM,C,[D1,D2..DN]). A 和 D 可以为空，但是 B 和 C 必须 非空。
2. 查询 单表
3. 查询没有 使用 group by 或 distinct
4. 查询 只使用了 index 中的 列
5. A1,,AK 上的 predicate(谓词) 必须是 equality predicate，且必须是 常量，包括 in 操作符
6. 查询必须是 conjunctive 查询 (连词查询)，即 多个and 连接的 or：
   (cond1(key_part1) OR cond2(key_part1)) AND (cond1(key_part2) OR ...) AND …
7. 必须有个 range condition 在C列上
8. D列上可以有 condition，D上的条件 必须 和 C上的 range condition 相结合。

使用 skip scan 时，explain 会有如下输出：
在 Extra 列中 的 using index for skip scan 表明 使用了 loose index skip scan access method
如果 索引可以被用于 skip scan, 索引会出现在 possible_keys 中。

使用 skip scan,会 在 优化器 trace output 中出现 下面形式的 skip scan 元素：

```
"skip_scan_range": {
  "type": "skip_scan",
  "index": index_used_for_skip_scan,
  "key_parts_used_for_access": [key_parts_used_for_access],
  "range": [range]
}
```

你还可能看到 best_skip_scan_summary 元素。如果 skip scan 被选为 最佳的 range access variant，会写一个 chosen_range_access_summary。如果 skip scan 被选为 overall(整体) 最佳 access method, 会存在 best_access_path 元素。

skip scan 的使用 取决于 optimizer_switch 系统变量的 skip_scan 标志的值。 默认这个 标志是 on。 设置为off 来关闭。

除了使用 optimizer_switch 系统变量 来控制 优化器 在 session 范围内 使用 skip scan 外，mysql 还支持 在每个语句 上 通过 optimizer hint 来 影响 优化器。

Range Optimization of Row Constructor Expressions

优化器 可以 应用 range scan access method 到 下面格式的 查询：

```
SELECT ... FROM t1 WHERE ( col_1, col_2 ) IN (( 'a', 'b' ), ( 'c', 'd' ));
```

以前，要使用 range scan 必须写成：

```
SELECT ... FROM t1 WHERE ( col_1 = 'a' AND col_2 = 'b' )
OR ( col_1 = 'c' AND col_2 = 'd' );
```

为了让 优化器 使用 range scan, 查询 必须 满足一下条件：

1. 只有 in 谓词 被用到， 不能有 not in
2. 在 in 左侧，行构造器 只包含 列引用 (row constructor contains only column references.)
3. 在 in 右侧，行构造器 只包含 运行时常量，它们 是在 执行期间 绑定到 常量的 文字 或 本地列 引用。
4. 在 in 右侧，有多个 行构造函数。

Limiting Memory Use for Range Optimization

要控制 range 优化器 的 可用内存， 使用 range_optimizer_max_mem_size 系统变量

0 意味着 无限制

>0，优化器 在 考虑 range access method 时 追踪 消耗的内存。 如果即将超过 限制，则放弃 range access method ，并考虑其他方法(包括 full table scan)。这可能不太理想。如果出现这种情况，会出现下面的警告 (N是 当前的 range_optimizer_max_mem_size 的值)：

```
Warning    3170    Memory capacity of N bytes for
                   'range_optimizer_max_mem_size' exceeded. Range
                   optimization was not done for this query.
```

对于update 和 delete 语句，如果 优化器 降级为 全表扫描 且 启用了 sql_safe_updates 系统变量， 则会是 error 而不是 warning。因为 实际上 没有 使用 任何 key 来 决定 哪些行 会修改。

对于超出 可用的 range优化内存 且 优化器 回退到 不太理想的 计划的 单个查询， 增加 range_optimizer_max_mem_size 可能会提高 性能。

要估计 处理 range 表达式 所需的内存量，请使用一下 准则：

像下面的 简单查询，它有一个 range access method 的 候选key，且 每个 谓词 通过 or 结合， 使用 大约 230 byte

```
SELECT COUNT(*) FROM t
WHERE a=1 OR a=2 OR a=3 OR .. . a=N;
```

。。应该是说 每个 or 要消耗 230byte

下面的查询，每个谓词 通过 and 连接，消耗 125 byte

```
SELECT COUNT(*) FROM t
WHERE a=1 AND b=1 AND c=1 ... N;
```

对于 in 的 查询：

```
SELECT COUNT(*) FROM t
WHERE a IN (1,2, ..., M) AND b IN (1,2, ..., N);
```

in中每个 文字 都算作 与 or 结合的 谓词， 如果有2个 in，那么 和 or组合的 谓词数量 是 每个 列表中 文字值数量的 乘积， 所以 上面的 谓词数是 M*N

---

https://dev.mysql.com/doc/refman/8.0/en/index-merge-optimization.html

Index Merge Optimization

index merge access method 检索 多个 range scan 的行，并 merge 成一个。
这个access method(存储方法) 只能合并 对 单表 的 index scan， 对 多表 的 scan 不能合并。
merge 可以生成 底层扫描的 并集，交集，交集的并集。

下面是使用 index merge 的 例子：

```
SELECT * FROM tbl_name WHERE key1 = 10 OR key2 = 20;

SELECT * FROM tbl_name
  WHERE (key1 = 10 OR key2 = 20) AND non_key = 30;

SELECT * FROM t1, t2
  WHERE (t1.key1 IN (1,2) OR t1.key2 LIKE 'value%')
  AND t2.key1 = t1.some_col;

SELECT * FROM t1, t2
  WHERE t1.key1 = 1
  AND (t2.key1 = t1.some_col OR t2.key2 = t1.some_col2);
```

。。 怎么看 index scan 是 单表 还是 多表。

index merge 优化算法 有下面的 已知限制

如果 你的query 有一个 带有深度 and/or嵌套 的复杂的 where子句， 且 MySQL 没有选择 最佳计划，请尝试使用 以下 转换：

```
(x AND y) OR z => (x OR z) AND (y OR z)
(x OR y) AND z => (x AND z) OR (y AND z)
```

index merge 不适用于 full-text 索引

在 explain 的输出中， index merge 方法 出现为 type 列的 index merge 。这种情况下，key 列 包含 已使用的index 的列表， key_len 包含 这些index 的 最长key part 的列表。

index merge access method 有多种算法，使用的算法 会 展现在 explain 的输出 的 extra 属性。

```
Using intersect(...)
Using union(...)
Using sort_union(...)
```

下面更详细地描述了 这些算法。优化器 根据 不同 选项的 可选 index merge算法 和其它访问方法 的 成本估计 进行选择

Index Merge Intersection Access Algorithm

这个 访问算法 适用于：where 子句 可以转化为 在 and连接的 不同key上的 多个 range condition，且 每个 condition 是以下之一：

下面这种形式的 N-part 表达式，它的 index 正好是 N part (即，所有 index part 被覆盖)：

key_part1 = const1 AND key_part2 = const2 ... AND key_partN = constN

InnoDB表的 主键上的 任意 range condition

例如：

```
SELECT * FROM innodb_table
  WHERE primary_key < 10 AND key_col1 = 20;

SELECT * FROM tbl_name
  WHERE key1_part1 = 1 AND key1_part2 = 2 AND key2 = 2;
```

index merge intersection 算法 对所有 使用的index 执行 同时扫描，并且 生成 它从 merged index scan 中检索到的 row 序列 的 交集。

如果 query 用到的 所有列 被 使用的索引 覆盖，则不会检索 full table row (这种情况下，explain 输出会有 在 Extra字段中的 Using index)。下面是这种查询的例子：

```
SELECT COUNT(*) FROM t1 WHERE key1 = 1 AND key2 = 1;
```

如果使用的索引 没有覆盖 query中的 所有列，则 仅当 满足所有的 使用的 key 的 range condition 时，才会 检索 full rows。

如果 其中一个 merge condition 是 对 InnoDB 表的 主键的 condition，则它不会用于 行检索，而是 用于 过滤 使用其它condition 检索到的行 。

Index Merge Union Access Algorithm

这个算法的 criteria 类似于 index merge intersection。 适用于： 表的 where 子句 转换为 or连接的 不同key 上的 多个 range condition，并且 每个 condition 是下面 之一：

一个 N-part 表达式，index 有 n个part (即，所有的 index part 都被覆盖)：

key_part1 = const1 AND key_part2 = const2 ... AND key_partN = constN

InnoDB 表 上的 主键 的 任意 range condition

index merge intersection 算法 适用的 condition

例子：

```
SELECT * FROM t1
  WHERE key1 = 1 OR key2 = 2 OR key3 = 3;

SELECT * FROM innodb_table
  WHERE (key1 = 1 AND key2 = 2)
     OR (key3 = 'foo' AND key4 = 'bar') AND key5 = 5;
```

Index Merge Sort-Union Access Algorithm
这个访问算法 适用于： where 子句 转换为 多个 or连接的 range condition 且 不适用 index merge union。
例子：

```
SELECT * FROM tbl_name
  WHERE key_col1 < 10 OR key_col2 < 20;

SELECT * FROM tbl_name
  WHERE (key_col1 > 10 OR key_col2 = 20) AND nonkey_col = 30;
```

sort-union 和 union 算法的区别是 ： sort-union 算法 必须先 fetch 所有row 的 row id，并在 返回 任何行 之前 sort 它们。

Influencing(影响) Index Merge Optimization
index merge 的使用 取决于 optimizer_switch 系统变量 的 index_merge, index_merge_intersection,index_merge_union,index_merge_sort_union 标记。 默认下，这些 都是 on。 通过设置为 off 可以关闭 指定算法。

除了使用 optimizer_switch 系统变量 来 控制 优化器 在 session 范围内 使用 index merge 算法外， MySQL 支持 在每个语句上 使用 optimizer hint 来 影响 优化器。

---

https://dev.mysql.com/doc/refman/8.0/en/hash-joins.html
8.2.1.4 Hash Join Optimization

默认下，MySQL(>=8.0.18) 尽可能 使用 hash join。
可以控制 是否 使用 hash join， 通过 BNL，NO_BNL 这2个 optimizer hint 之一， 或 设置 optimizer_switch 服务器系统变量的 block_nested_loop=on/off。
8.0.18 支持在 optimizer_switch 中设置 hash_join，以及 optimizer hint： HASH_JOIN, NO_HASH_JOIN。 在 > 8.0.18 的版本中 不再有效。

从8.0.18 开始，MySQL 对 每个join 有一个 等效join condition 且 没有index 可以被应用到 任何 join condition 上 的 查询， 使用 hash join.如：

```
SELECT *
    FROM t1
    JOIN t2
        ON t1.c1=t2.c1;
```

当有一个 或 多个 index 可以用于 单表 谓词时， hash join 也可以使用。

hash join 通常比 以前版本中 使用的 block nested loop 算法更快。 从8.0.20 开始，删除了 对 block nested loop 的支持。 服务器 在 以前使用 block nested loop 的 地方 使用 hash join。。。那你弄 开关干什么。。 只有 一个 hash join 了啊。 就只有 8.0.18 和 19 用到了。

在上面的例子 以及 本节 的 其余例子中， 我们假设 3个表 已经 创建：

```
CREATE TABLE t1 (c1 INT, c2 INT);
CREATE TABLE t2 (c1 INT, c2 INT);
CREATE TABLE t3 (c1 INT, c2 INT);
```

你可以通过 explain 来看到 hash join 被 使用了：

```
mysql> EXPLAIN
    -> SELECT * FROM t1
    ->     JOIN t2 ON t1.c1=t2.c1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: Using where; Using join buffer (hash join)
```

8.0.20之前，必须包含 format=tree 选项 来 查看 对于 给定的join 是否使用了 hash join。

explain analyze 还 显示 有关使用的 hash join 的 信息。

hash join 也用于 涉及 多个 join 的 查询，只要 每对表 至少有一个 join condition 时 equi-join， 例如：

```
SELECT * FROM t1
   JOIN t2 ON (t1.c1 = t2.c1 AND t1.c2 < t2.c2)
    JOIN t3 ON (t2.c1 = t3.c1);
```

在上面的例子中， 使用了 inner join， 任何不是 equi-join 的 condition 都会作为 join 执行后的 filter。(对于 outer join，如 left join， semijoin, antijoin, 它们被 printed as join 的一部分)。 这可以在 explain 的 输出中看到：

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->         ON (t1.c1 = t2.c1 AND t1.c2 < t2.c2)
    ->     JOIN t3
    ->         ON (t2.c1 = t3.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Inner hash join (t3.c1 = t1.c1)  (cost=1.05 rows=1)
    -> Table scan on t3  (cost=0.35 rows=1)
    -> Hash
        -> Filter: (t1.c2 < t2.c2)  (cost=0.70 rows=1)
            -> Inner hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
                -> Table scan on t2  (cost=0.35 rows=1)
                -> Hash
                    -> Table scan on t1  (cost=0.35 rows=1)
```

从上面也可以看到，多个 hash join 可以 用于 具有 多个 equi-join condition 中的 join。

8.0.20之前， hash join 不会被使用 如果 任意 一对 join的表 没有 至少 一个 equi-join condition， 并且使用 较慢的 block nested loop 。 8.0.20之后，上面的情况下，会使用 hash join. 如下所示

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT * FROM t1
    ->     JOIN t2 ON (t1.c1 = t2.c1)
    ->     JOIN t3 ON (t2.c1 < t3.c1)\G
*************************** 1. row ***************************
EXPLAIN: -> Filter: (t1.c1 < t3.c1)  (cost=1.05 rows=1)
    -> Inner hash join (no condition)  (cost=1.05 rows=1)
        -> Table scan on t3  (cost=0.35 rows=1)
        -> Hash
            -> Inner hash join (t2.c1 = t1.c1)  (cost=0.70 rows=1)
                -> Table scan on t2  (cost=0.35 rows=1)
                -> Hash
                    -> Table scan on t1  (cost=0.35 rows=1)
```

hash join 也适用于 笛卡尔积， 也就是说，当没有 指定 join condition时， 如下所示:

```
mysql> EXPLAIN FORMAT=TREE
    -> SELECT *
    ->     FROM t1
    ->     JOIN t2
    ->     WHERE t1.c2 > 50\G
```

```
*************************** 1. row ***************************
EXPLAIN: -> Inner hash join  (cost=0.70 rows=1)
    -> Table scan on t2  (cost=0.35 rows=1)
    -> Hash
        -> Filter: (t1.c2 > 50)  (cost=0.35 rows=1)
            -> Table scan on t1  (cost=0.35 rows=1)
```

>= 8.0.20, join 不再需要 包含 至少一个 equi-join 才会 使用 hash join。 这意味者 下面的查询 可以被 优化为 使用 hash join

inner non-equi-join:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 JOIN t2 ON t1.c1 < t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> Filter: (t1.c1 < t2.c1)  (cost=4.70 rows=12)
    -> Inner hash join (no condition)  (cost=4.70 rows=12)
        -> Table scan on t2  (cost=0.08 rows=6)
        -> Hash
            -> Table scan on t1  (cost=0.85 rows=6)
```

Semijoin

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1
    ->     WHERE t1.c1 IN (SELECT t2.c2 FROM t2)\G
*************************** 1. row ***************************
EXPLAIN: -> Nested loop inner join
    -> Filter: (t1.c1 is not null)  (cost=0.85 rows=6)
        -> Table scan on t1  (cost=0.85 rows=6)
    -> Single-row index lookup on <subquery2> using <auto_distinct_key> (c2
=t1.c1)
        -> Materialize with deduplication
            -> Filter: (t2.c2 is not null)  (cost=0.85 rows=6)
                -> Table scan on t2  (cost=0.85 rows=6)
```

Antijoin

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t2
    ->     WHERE NOT EXISTS (SELECT * FROM t1 WHERE t1.col1 = t2.col1)\G
*************************** 1. row ***************************
EXPLAIN: -> Nested loop antijoin
    -> Table scan on t2  (cost=0.85 rows=6)
    -> Single-row index lookup on <subquery2> using <auto_distinct_key> (c1
=t2.c1)
        -> Materialize with deduplication
            -> Filter: (t1.c1 is not null)  (cost=0.85 rows=6)
                -> Table scan on t1  (cost=0.85 rows=6)
```

left outer join

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> Left hash join (t2.c1 = t1.c1)  (cost=3.99 rows=36)
    -> Table scan on t1  (cost=0.85 rows=6)
    -> Hash
```

```
    -> Table scan on t2  (cost=0.14 rows=6)
```

Right outer join (可以看到 mysql 将 所有 right join 重写为 left join)

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 RIGHT JOIN t2 ON t1.c1 = t2.c1\G
*************************** 1. row ***************************
EXPLAIN: -> Left hash join (t1.c1 = t2.c1)  (cost=3.99 rows=36)
    -> Table scan on t2  (cost=0.85 rows=6)
    -> Hash
        -> Table scan on t1  (cost=0.14 rows=6)
```

默认下，>=8.0.18 尽可能使用hash join。 可以通过 BNL NO_BNL 优化器hint 来控制 是否使用 hash join

hash join 使用的 内存 可以通过 join_buffer_size 系统变量 来控制；hash join 不会使用 超过 这个 数量 的内存。
当 hash join 所需的内存超过这个数量时，mysql通过使用 磁盘上的文件 来处理这个问题。
如果发生这种情况，你需要直到： 如果 hash join 无法装入 内存，且 它创建的 文件 多于 open_file_limit 设置的值，则 join 可能失败。 为了避免这种问题，进行 下面的 任一 修改：

增加 join_buffer_size。
增加 open_files_limit

从8.0.18 开始，hash join 的 join buffer 是 增量分配的；因此，你可以将 join_buffer_size 设置得更高，而不必担心 小查询 使用 大量内存，但 outer join 会 申请 整个 buffer。在 >=8.0.20 中， hash join 也用于 outer join (包括 antijoin, semijoin)， 所以这个不再是 问题。

---

https://dev.mysql.com/doc/refman/8.0/en/engine-condition-pushdown-optimization.html

8.2.1.5 Engine Condition Pushdown Optimization

这个优化 提高了 非索引列 和 常量 之间 比较的 效率。
在这种情况下， condition 被 push down 到 storage engine 来进行评估。这个优化 只能由 NDB 存储引擎 使用。

对于ndb cluster ，这种优化 可以消除 发出查询的 mysql server 和 cluster的数据节点 间 传输 不匹配的 行的 消耗， 可以将 query 加速 5-10 倍。

假设 ndb cluster 表 定义如下：

```
CREATE TABLE t1 (
    a INT,
    b INT,
    KEY(a)
) ENGINE=NDB;
```

engine condition pushdown 可以用于如下的查询，其中包括 非index 列 和 常量 的比较。

```
SELECT a, b FROM t1 WHERE b = 10;
```

explain 的输出中 可以看到 engine condition pushdown

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE b = 10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 10
        Extra: Using where with pushed condition
```

然而，engine condition pushdown 不能用于下面的 query：

```
SELECT a,b FROM t1 WHERE a = 10;
```

原因是，a列 存在 index。(index access method 更效率，所以 优于 engine codition pushdown)

但 indexed 列 使用 <, > 对 常量进行比较时，也可以使用 engine condition pushdown：

```
mysql> EXPLAIN SELECT a, b FROM t1 WHERE a < 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: range
possible_keys: a
          key: a
      key_len: 5
          ref: NULL
         rows: 2
        Extra: Using where with pushed condition
```

其它支持 engine condition pushdown 的 比较是：

column [not] like pattern

pattern 必须是 包含要匹配的 模式的字符串文字

column is [not] null

column in (value_list)

value_list中的 每个 item 必须是 常量，文字值

column between constant1 and constant2

constant1, constant2 必须是 常量， 文字值

在前面列举的 所有情况中， condition 都可以转换为 列 和 常量 之间 的 一个或多个 直接比较。

默认开启了 engine condition pushdown。 要在server 启动时 禁止它，设置 optimizer_switch 系统变量的 engine_condition_pushdown 标记 为 off。 例如，在 my.cnf 文件中：

```
[mysqld]
```

```
optimizer_switch=engine_condition_pushdown=off
```

运行时，禁用 engine condition pushdown:

```
SET optimizer_switch='engine_condition_pushdown=off';
```

限制

engine condition pushdown 受制于下面的 限制：

1. 只有 NDB 存储引擎 支持 engine condition pushdown
2. <NDB 8.0.18，可以将 列 与 常量 或 只能推导出常量的表达式 进行比较。 >=NDB 8.0.18，列可以和 另一个列 相比较，只要 它们 的类型 完全相同，包括 相同的 signedness,length,character set, precision,scale (如果有的话)。
3. 比较的列 不能是 blob 或 text 类型。也不能是 json,bit,enum 列。
4. 要和 列 比较的 string 值 必须使用 和列 相同的 collation
5. 不直接支持 json；涉及多个表的 condition 被 尽可能 单独 push。 使用 explain 来确认 哪些 condition 被 实际 pushdown.

以前， engine condition pushdown was limited to terms referring to column values from the same table to which the condition was being pushed.
从NDB 8.0.16 开始，query plan 中 较早的表的 列值 也可以从 被push 的condition 中国 引用。这减少了 join处理期间 必须由 sql node 处理的 行数。过滤也可以在 LDM 线程中 并行，而不是在 单个 mysqld 进程中。 这可以大大提高查询 性能。

从NDB 8.0.20 开始，一个 使用 scan 的 outer join 能被 push ，如果 在同一 join nest中使用的 任何表 或 它所依赖的 join nest 的 任何表上 没有 不可推送的 condition。如果使用的 优化策略是 firstMatch，则 semijoin 也是如此。

在下面2种情况下，join算法 不能 和以前的表中的 引用列 进行组合

1. 当任何引用的 先前表 在 join buffer中。这种情况下， 从 scan-filtered table 中检索到的 每一行 都与 缓冲区中的 每一行 相匹配。 这 意味者，在 生成 scan filter 时，没有可以从中 获得 列值的 单个特定行。
2. 当 列 来自 已经push 的join 的 子操作中。 这是因为 在 scan filter 生成时 ，没有检索到 join 中 祖先操作 引用的 行。

NDB 8.0.27 开始，可以 下推 join 中 祖先表的 列， 前提时 它们满足 前面 列出的要求。比如：

```
mysql> EXPLAIN
    ->   SELECT * FROM t1 AS x
    ->   LEFT JOIN t1 AS y
    ->   ON x.a=0 AND y.b>=3\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: x
   partitions: p0,p1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
```

```
          rows: 4
      filtered: 100.00
         Extra: NULL
*************************** 2. row ***************************
            id: 1
  select_type: SIMPLE
         table: y
    partitions: p0,p1
          type: ALL
possible_keys: NULL
           key: NULL
       key_len: NULL
           ref: NULL
          rows: 4
      filtered: 100.00
         Extra: Using where; Using pushed condition (`test`.`y`.`b` >= 3); Using
join buffer (hash join)
2 rows in set, 2 warnings (0.00 sec)
```

---

https://dev.mysql.com/doc/refman/8.0/en/index-condition-pushdown-optimization.html

8.2.1.6 Index Condition Pushdown Optimization

index condition pushdown (ICP) 是一个优化，对于：MySQL 从一个 使用 index 的表中 检索数据。
没有ICP的话，存储引擎 遍历index 来 定位 base table 中 row 的位置，然后将 row 返回给 MySQL 服务器，MySQL 服务器来 评估 row 的 where 条件。
有ICP的话，如果 where 的condition 中有一部分 可以 只使用 index 中的列 就可以进行 评估，MySQL server 将这部分 condition 下推到 存储引擎。 存储引擎 会 通过 使用 索引条目 来 评估 推送的 index condition， 并且 只有在 满足这一条件时 才从表中 读取行。
ICP可以减少 存储引擎 访问 base table 的次数 和 MySQL 访问 存储引擎的次数。
。。base table != index

ICP 的适用性 收到 以下条件限制：

1. 当 需要 access full table rows 时， ICP 被用于 range,ref,eq_ref,ref_or_null 访问方法。
2. ICP 可以用于 InnoDB 和 MyISAM表， 包括 分区的 InnoDB 和 MyISAM 表
3. 对于 InnoDB 表，ICP 仅用于 secondary indexes。ICP 的目标是 降低 full-row read 的次数，从而减少IO操作。 对于 InnoDB clustered indexes(聚簇索引)，完整的记录已经读入 InnoDB buffer，ICP 不会减少 IO。
4. 在虚拟列 上的 二级索引 不支持 ICP。 InnoDB 可以在 虚拟列上 生成 二级索引。
5. 引用 子查询的 condition 不能 pushdown
6. 引用存储方法 的条件 不能 pushdown。 存储引擎 不能调用 存储方法。
7. triggered condition 不能 pushdown。
8. (>=8.0.30) condition 不能 被 pushdown 到 包含 对 系统变量的引用 的 派生表。

要了解 这个优化如何工作，首先考虑 不使用 ICP 的时候 index scan 如何 工作：

1. 获得下一行，首先通过 读取 index tuple, 然后 使用 index tuple 来定位 和 读取 full table row.
2. 测试 应用在这张表上的 where 条件。根据测试结果 接受 或拒绝 这个row

使用ICP，扫描会变成：

1. 获得 下一个 row 的 index tuple (不是 full table row)
2. 测试 应用到这个表 且 能只使用 index 的列 就可以 测试 的 where condition，如果不满足，回到第一步。
3. 如果 满足，使用 index tuple 定位 和 read full table row.
4. 测试 应用到这张表 的 剩余的 where condition。 根据测试结果，接受 或拒绝这个 row。

explain的输出 展示了 Using index condition 在 Extra 列中，当 ICP 被使用时。 不会展示 Using index 因为 当full table row 必须被读取时， 它并不适用。

假设一个表 包含了 关于人员和他们的地址 的信息， 并且 表 有一个 index，定义为 index (zipcode, lastname, firstname)。 如果我们知道 一个人的 zipcode，但不确定 last name，我们可以这样搜索：

```
SELECT * FROM people
  WHERE zipcode='95054'
  AND lastname LIKE '%etrunia%'
  AND address LIKE '%Main Street%';
```

MySQL 可以通过 zipcode='95054' 使用 index 来 scan。第二部分 (lastname like '%etrunia%') 不能 用来 限制 必须扫描的 row 的数量，因此 如果没有 ICP，则 查询 必须 检索 所有 zipcode='95054' 的人的 full table rows。
使用ICP后，MySQL 读取 整行前 检查了 lastname like '%etrunia%' 。 这避免了 读取 匹配 zipcode 但 不匹配 lastname 的 index tuple 的 full row。

默认，启用 ICP。 它可以通过 optimizer_switch 系统变量的 index_condition_pushdow 标记 控制：

```
SET optimizer_switch = 'index_condition_pushdown=off';
SET optimizer_switch = 'index_condition_pushdown=on';
```

---

https://dev.mysql.com/doc/refman/8.0/en/nested-loop-joins.html

8.2.1.7 Nested-Loop Join Algorithms

MySQL 使用 nested-loop 算法 或 它的变种 来 执行 表间 的 join

Nested-Loop Join Algorithm

一个简单的 nested-loop join (NLJ) 算法 一次 从 循环中的 第一个表 读取 行，将每一行 传递给 处理 join 中 下一个表的 nested loop。只要还有 要 join 的表，这个过程 就会 重复多次。

考虑 要执行 一个 3张表 (t1,t2,t3) 间的 join， 并且join type 如下：

Table	Join Type
t1	range
t2	ref
t3	ALL

如果使用 simple NLJ 算法，join 会被 如下处理：

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send to client
    }
  }
}
```

因为 NLJ 算法 一次 将一行 从 外循环 传递到 内循环，所以 它会 多处 读取 内循环中的 表。
。。行数的多少 都一样， 只有，列多的 放外面， 列少的 放里面。

Block Nested-Loop Join Algorithm
一个 block nested-loop (BNL) join 算法 采取 buffering 外层循环读取的 row 来 减少 必须读取的 内循环表 的 次数。
例如：如果 一次性读取 10 row 到 buffer，然后 传递 buffer 到 内循环， 内循环中 读取的 每行 可以 和 buffer中的 10 row 进行 比较，这可以减少 必须读取的 内循环表的 次数 一个数量级。

<MySQL 8.0.18，此算法 应用于 无法使用 index 的 equi-join。
>=8.0.18，这种情况下，使用 hash join 优化
>=8.0.20，MySQL 不再使用 block nested loop，并且在 之前使用 BNL join 算法的 所有地方 都使用 hash join。

。。。。 那就没什么用了。。就只有 18.19 2个版本 用来 BNL join。
。。。跳了

---
https://dev.mysql.com/doc/refman/8.0/en/nested-join-optimization.html

8.2.1.8 Nested Join Optimization

join 的语法 允许 nested join。

和SQL 标准 相比， table_factor 语法 得到了 扩展。 SQL标准 只接受 table_reference，而不是 一对括号 内的 列表。如果我们将 table_reference 列表中的 每个 逗号 视为 一个

inner join，那么这是 一个 保守的 扩展。

```
SELECT * FROM t1 LEFT JOIN (t2, t3, t4)
                 ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

等价于

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
                 ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

。。MySQL cross join是mysql中的一种连接方式,区别于内连接和外连接,对于cross join连接来说,其实使用的就是笛卡尔连接。

MySQL中，cross join 在语法上 等同于 inner join, 它们可以互换。
标准SQL 中， 它们不等价， inner join 需要和 on 一起使用， 否则 使用 cross join.

通常，如果只包含 inner join,括号 可以省略。考虑下面的连接：

```
t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
   ON t1.a=t2.a
```

在删除括号，并将 操作 分组到 左侧后， 该表达式 转换为 以下 表达式

```
(t1 LEFT JOIN t2 ON t1.a=t2.a) LEFT JOIN t3
    ON t2.b=t3.b OR t2.b IS NULL
```

然而，这2种 并不 等价 。 考虑 表 t1,t2,t3 有以下的状态

```
Table t1 contains rows (1), (2)
Table t2 contains row (1,101)
Table t3 contains row (101)
```

这种情况下，第一个表达式 返回 ： (1,1,101,101),(2,null,null,null)。 第二个表达式 返回 (1,1,101,101),(2,null,null,101)。

```
mysql> SELECT *
       FROM t1
            LEFT JOIN
            (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
            ON t1.a=t2.a;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL | NULL |
+------+------+------+------+

mysql> SELECT *
       FROM (t1 LEFT JOIN t2 ON t1.a=t2.a)
            LEFT JOIN t3
            ON t2.b=t3.b OR t2.b IS NULL;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
```

```
|    2 | NULL | NULL |  101 |
+------+------+------+------+
```

在下面的例子中， outer join 操作 和 inner join 一起使用：

```
t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
```

这个表达式 不能转为 下面的：

```
t1 LEFT JOIN t2 ON t1.a=t2.a, t3
```

这2个 表达式，会返回 不同的 result set

```
mysql> SELECT *
       FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL | NULL |
+------+------+------+------+

mysql> SELECT *
       FROM t1 LEFT JOIN t2 ON t1.a=t2.a, t3;
+------+------+------+------+
| a    | a    | b    | b    |
+------+------+------+------+
|    1 |    1 |  101 |  101 |
|    2 | NULL | NULL |  101 |
+------+------+------+------+
```

因此，如果 我们在 带有 outer join 的表达式中 省略 括号，可能会 修改 原始 表达式的结果集。

更准确地说， 我们不能 忽略 left outer join 的 右操作数 ，和 right outer join 的 左操作数 中的 括号。 换句话说， 我们不能忽略 outer join 操作的 inner table expression 的 括号。 可以 忽略 其他 操作数 (outer table 的 操作数) 的括号。

下面的表达式：

```
(t1,t2) LEFT JOIN t3 ON P(t2.b,t3.b)
```

对于 任何表 t1,t2,t3 和 属性 t2.b 和 t3.b 上的 任何 条件 P， 等效于 下面的表达式：

```
t1, t2 LEFT JOIN t3 ON P(t2.b,t3.b)
```

每当 join 表达式 (joined_table) 中 join 操作的 执行顺序 不是 从左到右的， 我们就 讨论 nested join。 考虑以下 查询：

```
SELECT * FROM t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b) ON t1.a=t2.a
  WHERE t1.a > 1

SELECT * FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
  WHERE (t2.b=t3.b OR t2.b IS NULL) AND t1.a > 1
```

这些查询 被认为 包含这些 nested join

```
t2 LEFT JOIN t3 ON t2.b=t3.b
```

t2, t3

在第一个查询中，nested join 由 left join 构成。 第二个查询中，由 inner join 组成。

在第一个查询中， 括号可以省略：join表达式的 语法结构 规定了 join操作的 相同的 执行顺序。 第二个查询中， 括号不能省略，尽管 此处的 join 表达式 可以在 没有 括号的 情况下 依然明确 结束。 在我们的 扩展语法中， 第二个查询的 (t2,t3) 的括号 是必须的， 尽管理论上 可以在没有它们的情况下解析查询： 我们仍然 会有 查询的 明确的语法结构，因为 left join 和 on 扮演了 表达式 (t2,t3) 的 左右分隔符。

上面的例子说明了以下几点：

1. 对于只涉及 inner join 的 join 表达式，可以删除 括号，并 从左 到右 计算 join。
2. 通常，对于 outer join 或 对于 outer join和inner join混合的情况， 情况并非如此，删除括号 可能改变结果。

具有 nested outer join 的 查询 以 和 具有inner join 的查询 相同的 pipeline 方式执行。 更准确地说， 利用了 nested-loop join 算法变体。
回想一下 neste-loop join 执行查询时的 算法。假设 对3个表 有如下的 join 查询

```
SELECT * FROM T1 INNER JOIN T2 ON P1(T1,T2)
                 INNER JOIN T3 ON P2(T2,T3)
  WHERE P(T1,T2,T3)
```

这里，P1(T1,T2) 和 P2(T2,T3) 是 一些 (表达式上的) join condition，而 P(T1,T2,T3) 是表 T1，T2，T3 列的条件。

nested-loop join 算法 将以 下面的方式 执行这个查询：

```
FOR each row t1 in T1 {
  FOR each row t2 in T2 such that P1(t1,t2) {
    FOR each row t3 in T3 such that P2(t2,t3) {
      IF P(t1,t2,t3) {
         t:=t1||t2||t3; OUTPUT t;
      }
    }
  }
}
```

符号 t1||t2||t3 表示 通过 连接 行t1,t2,t3 的列 构成的 行。
下面的某些例子中，表名是null 意味着 该表的 每一列 都是 使用 null的 行。例如 t1||t2||null 表示 连接 行t1和t2 的 列 构造的 行，并且 t3 的每一列 都是 null。 这样的行被称为 null-complemented。

考虑下面的 带有 nested outer join 的 查询：

```
SELECT * FROM T1 LEFT JOIN
               (T2 LEFT JOIN T3 ON P2(T2,T3))
               ON P1(T1,T2)
  WHERE P(T1,T2,T3)
```

对于这个查询，修改 nested-loop 模式 以获得：

```
FOR each row t1 in T1 {
```

```
    BOOL f1:=FALSE;
    FOR each row t2 in T2 such that P1(t1,t2) {
      BOOL f2:=FALSE;
      FOR each row t3 in T3 such that P2(t2,t3) {
        IF P(t1,t2,t3) {
          t:=t1||t2||t3; OUTPUT t;
        }
        f2=TRUE;
        f1=TRUE;
      }
      IF (!f2) {
        IF P(t1,t2,NULL) {
          t:=t1||t2||NULL; OUTPUT t;
        }
        f1=TRUE;
      }
    }
    IF (!f1) {
      IF P(t1,NULL,NULL) {
        t:=t1||NULL||NULL; OUTPUT t;
      }
    }
  }
```

通常，对于 outer join 操作 中 第一个 inner table 的 任何 nested loop，都会 引入 一个 flag，这个flag 在 循环之前 关闭 并在 loop 后 检查。
flag 被打开，当 外部表中的当前行 与 表示内部操作数的 表匹配时。
如果在 循环结束时， 这个flag 依然是 关闭， 则没有 为 外部表 的 当前行 找到匹配项。
在这种情况下，该行由 内部表 列的 null值补充。 结果行 被传递给 输出的 最终结果 或 下一个 nested loop， 但前提是 该行 满足 所有 嵌入式的 outer join 的 join condition。
。。。。

在例子中，嵌入了由 以下表达式 表示的 outer join table。

```
(T2 LEFT JOIN T3 ON P2(T2,T3))
```

对于 带有 inner join 的 查询， optimizer 可以选择 不同的 嵌套循环的 顺序：

```
  FOR each row t3 in T3 {
    FOR each row t2 in T2 such that P2(t2,t3) {
      FOR each row t1 in T1 such that P1(t1,t2) {
        IF P(t1,t2,t3) {
          t:=t1||t2||t3; OUTPUT t;
        }
      }
    }
  }
```

对于带有outer join 的 query，优化器 只能选择 ：先 outer table 的loop，然后再 inner table 的loop。 因此，对于我们的 outer join 的 query， 只有 一种 嵌套顺序 可选。
对于下面的 查询， 优化器 评估了 2个 不同的 嵌套。这2种嵌套中，T1 必须在 外循环中 处

理，因为 它用于 外连接。 T2和T3 用于 内部连接，因此 必须在 内loop 中 处理 join。 但是，由于 join 是 inner join ， T2 和 T3 能 任意 顺序 处理。

```
SELECT * T1 LEFT JOIN (T2,T3) ON P1(T1,T2) AND P2(T1,T3)
  WHERE P(T1,T2,T3)
```

一种评估是 先 t2 再 t3

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t2 in T2 such that P1(t1,t2) {
    FOR each row t3 in T3 such that P2(t1,t3) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f1:=TRUE
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

另一种是 先 t3 再 t2

```
FOR each row t1 in T1 {
  BOOL f1:=FALSE;
  FOR each row t3 in T3 such that P2(t1,t3) {
    FOR each row t2 in T2 such that P1(t1,t2) {
      IF P(t1,t2,t3) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f1:=TRUE
    }
  }
  IF (!f1) {
    IF P(t1,NULL,NULL) {
      t:=t1||NULL||NULL; OUTPUT t;
    }
  }
}
```

在 讨论 inner join 的 nested-loop 算法时， 我们省略了一些 对 查询 执行性能 影响很大的 细节。 我们没有提到 所谓的 condition pushdown 。假设 我们的 where condition P(T1,T2,T3) 可以用 conjunctive formula 表示：

```
P(T1,T2,T2) = C1(T1) AND C2(T2) AND C3(T3).
```

这种情况下，MySQL 实际上使用 以下嵌套循环 算法 来执行 带有 inner join 的 查询：

```
FOR each row t1 in T1 such that C1(t1) {
  FOR each row t2 in T2 such that P1(t1,t2) AND C2(t2)  {
```

```
        FOR each row t3 in T3 such that P2(t2,t3) AND C3(t3) {
          IF P(t1,t2,t3) {
             t:=t1||t2||t3; OUTPUT t;
          }
        }
      }
    }
```

你会看到 C1(T1),C2(T2),C3(T3) 中 每个 conjunct 都从 最内层 loop 推到 可以对其进行评估的 最外层 loop。 如果 C1(T1) 是一个 非常严格的 条件，则 这个 condition pushdown 可能 大大减少 从 T1 传递到 内部循环的 row。 因此 查询速度 会大大提高。

对于 带有 outer join的 查询，只有 在发现 外部表中的 当前行 和 内部表 匹配后，才检查 where 条件。因此， 从 内部loop push 出来 condition 不能 直接应用于 outer join 的查询中。 在这里，我们必须 引入 condition pushed-down predicate，这些谓词 由 遇到匹配时 打开的 标志保护。。。。。

回想 这个 带有 outer join 的 例子

P(T1,T2,T3)=C1(T1) AND C(T2) AND C3(T3)

例如，使用 guarded pushed-down condition 的 nested-loop algorithm 看起来：

```
FOR each row t1 in T1 such that C1(t1) {
  BOOL f1:=FALSE;
  FOR each row t2 in T2
      such that P1(t1,t2) AND (f1?C2(t2):TRUE) {
    BOOL f2:=FALSE;
    FOR each row t3 in T3
        such that P2(t2,t3) AND (f1&&f2?C3(t3):TRUE) {
      IF (f1&&f2?TRUE:(C2(t2) AND C3(t3))) {
        t:=t1||t2||t3; OUTPUT t;
      }
      f2=TRUE;
      f1=TRUE;
    }
    IF (!f2) {
      IF (f1?TRUE:C2(t2) && P(t1,t2,NULL)) {
        t:=t1||t2||NULL; OUTPUT t;
      }
      f1=TRUE;
    }
  }
  IF (!f1 && P(t1,NULL,NULL)) {
      t:=t1||NULL||NULL; OUTPUT t;
  }
}
```

通常，pushed-down predicate 能从 join condition 中提取，如 P1(T1,T2)和 P2(T2,T3)。这种情况下，下推谓词 也由 一个 flag 保护，该flag 阻止 相应的 outer join 操作 生成的 null 补充行的 谓词 的check。

如果它 是由 where condition 的 predicate 诱导的， 则禁止在 同一个 nested join 中 通过 key 来 进行 从一个inner table 到 另一个 inner table 的 访问。

---

https://dev.mysql.com/doc/refman/8.0/en/outer-join-optimization.html

8.2.1.9 Outer Join Optimization

outer join 包含 left join 和 right join

MySQL 实现 一个 A Left Join B 的 join_specification 如下：

1. 表B 设置为 依赖于 表A 以及 A 所依赖的 所有表
2. 表A 设置为 依赖 left join condition 中 使用的 所有 表 (B除外)
3. left join condition 用来 决定 如何 从 B中检索 row。(换句话说，不使用 where子句的 任何 condition)
4. 执行所有 标准 join 优化，除了 一个表 总是在它依赖的 所有表之后 被读取 的情况。如果存在 循环 依赖，则会发生错误。
5. 执行所有 标准的 where 优化
6. 如果 A 中存在 和 where 子句匹配的 行，但 B中 没有 与 on 匹配的 行，则会生成 额外的 B 行，其中所有列 都被设置为 null
7. 如果 你使用 left join 查询 某个表中 不存在的 行，并且 你有以下测试：where中有 col_name is null，但是 col_name 这个列 被声明为 not null， MySQL停止搜索 更多的 row (对于特定的 组合key) 在找到 与 left join 匹配的 行之后。

right join 和 left join 实现类似，但是 表的 角色 颠倒。 right join 被 转化为 等价的 left join。

对于left join，如果生成 null 行的 where condition 始终 是 false (。。就是不会生成 null 行)， 则将 left join 改为 inner join。。 例如，如果 t2.column1 为null，则下面查询中 的 where 子句 将为 false：

```
SELECT * FROM t1 LEFT JOIN t2 ON (column1) WHERE t2.column2=5;
```

因此，可以安全地 转为 inner join:

```
SELECT * FROM t1, t2 WHERE t2.column2=5 AND t1.column1=t2.column1;
```

>= MySQL 8.0.14，在准备过程中，由 恒定字面 表达式 产生的 条件 被删除， 而不是在 优化阶段，此时已经简化了 join。 较早的 去除 琐碎条件(trivial condition) 允许 优化器 将 外连接 转换为 内连接； 这可能 会导致 改进的 查询计划 在 where 子句中 包含 琐碎条件的 outer join， 比如：

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 WHERE condition_2 OR 0 = 1
```

优化器 在 准备 过程中 看到 0=1 始终false， 所以多余的，会删除：

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 where condition_2
```

现在 优化器 可以 将查询 重写为 内部连接：

```
SELECT * FROM t1 JOIN t2 WHERE condition_1 AND condition_2
```

现在，优化器 可以在 表T1 之前使用 表 T2， 如果这样 会 导致 更好ode 查询计划。要提供 关于 table join order 的hint， 查看 8.9.3 的 Optimizer Hints。或者 使用 STRAIGHT_JOIN。，但是 STRAIGHT_JOIN 可能阻止 使用 index，因为 它 禁用了 semijoin 转换。

---

https://dev.mysql.com/doc/refman/8.0/en/outer-join-simplification.html

8.2.1.10 Outer Join Simplification
许多情况下，查询的 from子句中的 table expression 被简化了。

在解析器(parser)阶段， 具有 right outer join 操作的 查询 被转化为 仅包含 left join 操作 的 等效查询。 一般情况下，转换 会对如下的 right join 使用：

```
(T1, ...) RIGHT JOIN (T2, ...) ON P(T1, ..., T2, ...)
```

变成等效的 left join

```
(T2, ...) LEFT JOIN (T1, ...) ON P(T1, ..., T2, ...)
```

所有 类似 T1 inner join T2 on P(T1,T2) 的 inner join 都被替换为 T1,T2 P(T1,T2) 作为 where condition (或 嵌入join 的 join condition， 如果有的话)

当优化器 评估 outer join 操作的 plan 时， 它只考虑 对于每个这样的操作，outer table 在 inner table 之前 被 access 的 计划。 优化器的 选择 是有限的， 因为只有这样的计划， 才能 使用 nested-loop 算法 执行 outer join.

考虑下面 这种形式的 query，其中 R(T2) 极大 缩小了 表 T2 中 匹配行的 数量：

```
SELECT * T1 FROM T1
  LEFT JOIN T2 ON P1(T1,T2)
  WHERE P(T1,T2) AND R(T2)
```

如果 执行上面的 query，优化器 别无选择，只能 先访问 限制较少的 表 T1， 然后 访问 限制多的 表T2， 这可能会产生 低效的 执行计划。

相反，如果 where condition 是 null-rejected(拒绝为空)，MySQL 会将 查询转换为 没有 外连接操作 的查询。(也就是说，它将外连接 转换为 inner join)。一个 condition 被称为 对于 outer join 操作 的 null-rejected，如果 它 对于 为操作生成 的 任何 null-complemented 行 推导 出 false 或 unknown

因此，对于这个 outer join

```
T1 LEFT JOIN T2 ON T1.A=T2.A
```

下面的condition 是 null-rejected，因为 它们对于 任何 null-complemented row (T2列设置为 NULL) 都不能为 真

```
T2.B IS NOT NULL
T2.B > 3
T2.C <= T1.C
T2.B < 2 OR T2.C > 1
```

下面的不是 null-rejected ，因为 它们 对于 null-complemented row 可能是 true：

T2.B IS NULL

T1.B < 3 OR T2.B IS NOT NULL

T1.B < 3 OR T2.B > 3

检查 outer join 操作的 condition 是否是 null-rejected 的 一般规则是：

1. 它是 A is not null 的 形式，A是任意 inner table 的 属性。
2. 它是 一个 包含 对 inner table 引用的 谓词，当其参数 之一 为null 时，谓词计算结果是 unknown
3. 它是一个 conjunction 包含了 一个 null-rejected condition 作为 conjunct
4. 它是 多个 null-rejected condition 的 disjunction

。。合取词conjunction (and) ；；；； 析取词disjunction or 。。。

一个condition 在 一个query中 可以是 对于outer join操作 是 null-rejected， 对于另一个(..query?不，是 outer join) 可能不是 null-rejuected 的。

下面的 where condition 对于 第二个 outer join 是 null-rejected ， 但是 对于 第一个不是 null-rejected。

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 LEFT JOIN T3 ON T3.B=T1.B
  WHERE T3.C > 0
```

如果 where condition 对于 query 中的 某个 outer join 是 null-rejected 的， 那么 这个 outer join 被替换为 inner join

例如，在上面的 query 中，第二个 outer join 是 null-rejected, 所以可以用 inner join 替换。

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 INNER JOIN T3 ON T3.B=T1.B
  WHERE T3.C > 0
```

对于原始query，优化器 仅评估 与 单表访问顺序 T1,T2,T3 兼容的计划。 对于 重写的 查询，它还考虑 访问顺序 T3,T1,T2

一个 outer join 的转换 可能触发 另一个 outer join 的转换， 所以，query：

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 LEFT JOIN T3 ON T3.B=T2.B
  WHERE T3.C > 0
```

第一次转为：

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
                 INNER JOIN T3 ON T3.B=T2.B
  WHERE T3.C > 0
```

这个等价于：

```
SELECT * FROM (T1 LEFT JOIN T2 ON T2.A=T1.A), T3
  WHERE T3.C > 0 AND T3.B=T2.B
```

剩余的 outer join 也可以被 替换为 inner join， 因为 条件 T3.B=T2.B 是 null-rejected。这会 导致 查询 没有 outer join

```
SELECT * FROM (T1 INNER JOIN T2 ON T2.A=T1.A), T3
  WHERE T3.C > 0 AND T3.B=T2.B
```

有时，优化器 成功替换 embedded outer join 操作，但 无法 转换 embedding outer join。下面的查询：

```
SELECT * FROM T1 LEFT JOIN
              (T2 LEFT JOIN T3 ON T3.B=T2.B)
              ON T2.A=T1.A
  WHERE T3.C > 0
```

被转换为

```
SELECT * FROM T1 LEFT JOIN
              (T2 INNER JOIN T3 ON T3.B=T2.B)
              ON T2.A=T1.A
  WHERE T3.C > 0
```

只能将其 重写为 依然 包含 embedding outer join 的 形式

```
SELECT * FROM T1 LEFT JOIN
              (T2,T3)
              ON (T2.A=T1.A AND T3.B=T2.B)
  WHERE T3.C > 0
```

任何企图 转换 query中 embedded outer join 的尝试 都必须 考虑 embedding outer join 的 join condition 和 where condition。
在这个query中， 对于 outer join ，where condition 不是 null-rejected 的， 但是 outer join 的 join condition ： T2.A=T1.A and T3.C=T1.C 是null-rejected。

```
SELECT * FROM T1 LEFT JOIN
              (T2 LEFT JOIN T3 ON T3.B=T2.B)
              ON T2.A=T1.A AND T3.C=T1.C
  WHERE T3.D > 0 OR T1.D > 0
```

因此，这个query 可以转换为

```
SELECT * FROM T1 LEFT JOIN
              (T2, T3)
              ON T2.A=T1.A AND T3.C=T1.C AND T3.B=T2.B
  WHERE T3.D > 0 OR T1.D > 0
```

---

https://dev.mysql.com/doc/refman/8.0/en/mrr-optimization.html

8.2.1.11 Multi-Range Read Optimization
当表很大 且 没有存储在 存储引擎 的缓存中时，使用 二级索引上的 范围扫描 可能导致 对基表的 许多随机 磁盘访问。
借助 disk sweep Multi-Range Read(MRR) 优化，MySQL 尝试 通过 首先 仅扫描index 并收集相关 row 的key 来减少 range scan 的 randon disk access 的次数。然后 对 key 进行排

序，最后使用 primary key 的顺序 从 base table 检索 row。
disk-sweep MRR 的动机 是 减少 random disk access 的 次数， 实现 对 base table 的 更顺序的扫描 (more sequential scan)。

multi-range read 优化提供了下面的 好处：

1. MRR 允许 基于 index tuple 按照顺序 访问 data row，而不是 随机访问。服务器获得满足 query condition的 index tuple集合，根据 data row ID 进行排序，使用 排序后的 tuple 来 按顺序 检索 data row。 这使得 数据访问 更高效，成本更低。
2. MRR 支持 对 通过index tuple进行row access操作的 key access 请求 进行 批处理，例如 range index scan 和 equi-join 使用 index 作为 join 属性。MRR遍历 一系列 index range 来 获得 合格的 index tuple。随着这些结果的 积累，它们用于访问 相应的 data row。 在开始读取数据前 不必获得 所有的 index tuple

MRR优化 不支持 虚拟列 上创建的 二级索引。 InnoDB 可以在 虚拟列上 生成 二级索引。

以下场景说明了 MRR 优化 何时可以发挥优势：
场景A： MRR 用在 InnoDB 和 MyISAM 表上 进行 index range scan 和 equi-join 操作。

1. index tuple 的一部分 在 buffer 中 累积
2. buffer中的 index tuple 按照 data row ID 排序
3. 根据 排序后的 index tuple sequence 访问 data row。

场景B： MRR 用在 NDB 表上 进行 multi-range index scan 或 通过 attribute 来执行 equi-join

1. range的一部分(可能是single-key range)，累积在 提交query的 central node 的 buffer中。
2. range 被发送到 执行node 来 access data row
3. 访问到的 row 被 打包 并 发回 central node
4. 接收到的 包 被放到 buffer中
5. 从buffer 中读取 data row

当MRR 被使用时， explain 的output 的 Extra 列 显示 Using MRR。

InnoDB 和 MyISAM 不会使用 MRR ，如果 不需要 访问 full table row 来生成结果。 如果 结果可以 完全 基于 index tuple (通过 覆盖索引(convering index)) 中的信息 产生， 那么 MRR 没有任何好处。

2个 optimizer_switch 系统变量flag 提供了 使用 MRR 优化的 接口。 mrr 标记 控制 是否启用 MRR。 如果 mrr 是 on， mrr_cost_based 标记 控制 是否让优化器 做出一个 基于 成本的 选择： on的话会 基于成本 选择 使用 或 不使用MRR， off 的话 能用MRR的地方就 使用MRR。 默认下 mrr 是 on， mrr_cost_based 是 on。

对于MRR， 存储引擎 使用 read_md_buffer_size 系统变量的 值 作为 它可以为buffer 分配多少 内存的 指南。 引擎最多 使用 read_md_buffer_size 字节，并确认 单次处理 的 range 的数量。

---

https://dev.mysql.com/doc/refman/8.0/en/bnl-bka-optimization.html

8.2.1.12 Block Nested-Loop and Batched Key Access Joins

MySQL中， batched key access (BKA) join 算法 使用了 joined table 的 index access 和 join buffer。BKA算法支持 inner join,outer join, semijoin，包括 nested outer join。BKA的好处包括 通过更有效的 表扫描 来提高 join 性能。 此外，以前仅 用于 inner join de block nested-loop (BNL) 算法 得到了 扩展，可以用于 outer join，semijoin，包括 nested outer join。
。。。BNL 不是 被 hash 替代了吗？

下面讨论，原始BNL算法，扩展BNL算法，BKA算法 的 join buffer 的管理。

Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms
MySQL不仅可以 使用 join buffer 来执行 没有索引访问内部表的 inner join, 还可以执行 在 subquery flatten 后 出现的 outer join 和 semijoin。 此外，当对内部表 进行索引访问时，可以有效地 使用 join buffer。

join buffer管理者 在存储 感兴趣的行的值时 更有效地 利用 join buffer 空间。 如果 列的值为null，则不会在 buffer 分配额外的字节，并且 为任何 varchar 类型的 列 分配 最少字节数。

join buffer管理器 支持 2种类型的 buffer，常规和增量 (regular， incremental)。 支持 join buffer B1 用于 join 表t1 和 t2，并且这个操作的结果 使用buffer B2 来和 表t3 进行 join：

1. regular join buffer 包含 来自 每个join 操作数 的 column。 如果B2 是 regular j-b， 则放入 B2 的 每row r 都由 来自 B1的 row r1 的列 和 来自 表t3的 匹配行 r2 的 interesting 列 组成
2. incremental j-b 仅包含 第二个 join 操作数(表) 产生的row 中的 列。 也就是说， 它对 第一个 操作数 缓冲区 的 row 进行 递增。如果 B2 是 incremental join buffer，它包含 r2 中的 interesting 列 以及 从 B1 到 行 r1 的 link

增量j-b 始终是 相对于 前一个 join 操作的 j-b 的 增量， 所以 第一个 join操作 必须是 常规 j-b。 在上面的例子中， 用于 join t1 和 t2 的 j-b B1 必须是 常规缓冲区。

join操作的 增量buffer 的每一行 只包含 要join的 表中 感兴趣的行。 这些列 通过 对第一个 连接 操作数 生成的 表中 匹配行的 感兴趣列 的 引用 进行扩充。增量join-buffer 中的几行 可以 ref 到 先前j-b 的同一行。

增量j-b 可以减少 从 之前的join操作使用的 buffer 中 复制 列的 频率。 这节省了 buffer 空间，因为 在一般情况下， 第一个join 产生的 row 可以和 第二个 join的 表的 数行 匹配。没有必要 从第一个操作数 中复制 多行。 由于 减少了复制的时间，增量j-b 还可以节省处理时间。

MySQL 8.0 中，optimizer_switch系统变量的 block_nested_loop 标记 如下工作：

1. <MySQL 8.0.20, 它控制 优化器 怎么 使用 block nested loop join 算法
2. >=8.0.18，也控制了 hash join 的使用
3. >=8.0.20， 只控制 hash join， block nested loop 算法不再支持。

batched_key_access 控制 优化器 怎么使用 batched key access join 算法

默认，block_nested_loop 是 on ， batched_key_access 是 off。
可以使用 optimizer hint。(。。是一个 link。。 最后一个是 optimizer hint 大章节《8.9.3 Optimizer Hints》)

Block Nested-Loop Algorithm for Outer Joins and Semijoins
MySQL BNL 算法的 原始实现 被 扩展 以 支持 outer join 和 semijoin 操作 (后来 被 hash join 算法 取代)。

当 使用 join buffer 执行 这些 操作时， 放入buffer 中的 每行 都提供一个 match flag。

如果 outer join 在执行时 使用了 join buffer， 则 检查 第二个操作数 生成的 表的 每行 是否 和 join buffer 中的 每行 匹配。 当找到匹配时，将 形成 一个新的 扩展行(原始行 加上 来自 第二个操作数 的 列) 并发给 剩余的 join操作 来进一步扩展。 此外，启用 buffer中的 匹配的row 的 match flag。 在检测完 要join的 表的 所有 row 之后， 扫描 join buffer。 buffer中 每个 没有 启用 match flag 的 row 都 被 null complements 扩展 (第二个操作数的 每列 都是 null 值) 然后 发送给 剩余的 join 操作 来进一步扩展。

MySQL 8.0 中，optimizer_switch系统变量的 block_nested_loop 标记 如下工作：
1. <MySQL 8.0.20， 它控制 优化器 怎么 使用 block nested loop join 算法
2. >=8.0.18，也控制了 hash join 的使用
3. >=8.0.20， 只控制 hash join， block nested loop 算法不再支持。

可以使用 optimizer hint.

在 explain 的输出中，当 extra 的值 包含 Using join buffer (Blocked Nested Loop) 且 type 的值是 ALL,index,range， 表示 对表 使用了 BNL。

Batched Key Access Joins
MySQL 实现了一种 join 表的方式，称为 Batched Key Access (BKA) join 算法。
BKA用于：当 对 第二个join操作数 生成的 表 进行 index access 时。
和 BNL join算法一样， BKA join 算法 使用 join-buffer 来 累积 join操作的第一个操作数 产生的 行的 感兴趣的 列。 然后 BKA算法 为buffer 中 所有行 构建key 来 访问 要join 的 表，并将这些key 批量提交到 数据库引擎 来进行 index 查找。 这些 key 通过 Multi-range read (MRR) 接口 来提交给引擎。 提交key 后， MRR 引擎函数 以最佳的方式 在index 中 执行 查找，通过这些key 找到 joined 表的 row，并 开始为 BKA join 算法 提供 匹配的 row。每个匹配的row 都与join buffer 中 row 的引用 相结合

使用BKA 时，join_buffer_size 的值 定义了 每个 发给 存储引擎的 请求中 key的 batch 的 大小 (。。就是一批几条key)。 这个值越大， 对join操作 的 右表 的 顺序访问 越多， 这可以 显著提高性能。

要使用BKA， optimizer_switch 系统变量的 batched_key_access 标志 必须设置为on。BKA 使用 MRR，所以 mrr 标志 也必须 on。 目前，MRR的 成本估计 过于 悲观。因此，还需要 关闭 mrr_cost_based 才能使用 BKA。 下面设置 启用 BKA

```
mysql> SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

MRR函数有 2种 执行情况：

1. 第一个场景用于 传统的 disk-based 存储引擎 如InnoDB，MyISAM，对于这些引擎，通常来自 join buffer 的 所有row 的 key 会 立刻 被提交给 MRR。 特定于引擎的 MRR函数 对 提交的key 执行 index lookup，从中 获得 row ID (或 主键)，然后 根据 BKA算法 的request 获取 所有这些选定的 row id的 row。 每个row 都返回一个 关联的 引用，该引用 允许访问 join buffer中的 匹配的row。 MRR以最佳方式 获得row：row 以 row id (主键) 的顺序 被 fetch。 这提高了 性能，因为读取是按 磁盘顺序的 ，而不是 随机顺序的。
2. 第二种场景，用于 NDB 等 远程存储引擎。来自 join buffer 的 row 的部分 的 key 及其关联 打包后，由 MySQL server (SQL NODE) 发送到 MySQL Cluster data node。 in return, sql node 收到 一个或多个 包，包中是 匹配的row 和 响应的关联。 BKA join 算法 采用这些row 并构建 新的 joined row。 然后 将新的 key 集合 发送到 data node，然后 用 data node返回的 包 汇总的 row 构建 新的 joined row。这些过程 持续进行 直到 join buffer 中的 最后一个 key 被送到 data node，并且 SQL node 收到并 join 了 这些key 匹配的 所有行。 这提高了 性能，因为 SQL node 向 data node 发送的 key 包 越少越好， 这意味着 它 和 data node 之间的 为了join 而 进行的 通信往返 次数 就越少。

在第一种情况下，部分 join buffer 空间被 保留 来 保存 由index loop选择的，并会作为参数 发给 MRR 函数 的 rowID（primary key）。

没有特殊的 buffer 来存储 为了 join buffer 中的 row 而构建的 key。相反，为 buffer 中下一个 row 构建 key 的 函数 被作为 参数 传递给 MRR 函数。

在explain的输出中，当 Extra 包含 Using join buffer (Batched Key Access) 并且 type 值是 ref 或 eq_ref，则表明 使用了 BKA。

Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms
除了使用 optimizer_switch 系统变量 来控制 BNL 和 BKA 在 会话范围内的使用，还支持 在每个语句上 使用 optimizer hint。
要使用 BNL 或 BKA hint 来 为 outer join 的 任何 inner table 启用 join buffer，必须为 outer join 的 所有 inner table 启用join buffering 。

---
https://dev.mysql.com/doc/refman/8.0/en/condition-filtering.html

8.2.1.13 Condition Filtering

在 join 处理中，prefix row 是那些 在join 中 从 一个表中 被传递到 下一个表 的那些行。通常， 优化器 会尝试 将 具有 low prefix count 的 表 放在 join 的 早期，以防止 row combination 的数量 过快地 增长。如果优化器 可以使用 关于 从一个表中 选择 并 传递给下一个表的 row 的 condition 的 信息， 它可以 更准确地 计算 行估计 并选择 最佳 执行计划。

在没有 condition filtering 的时候，表的 prefix row count 基于 where 子句 根据 优化器 选择的 access method 估计行数。 condition filtering 允许 优化器 使用 access method 没有考虑的 where子句中 其他相关条件。 例如，即使有一种 基于 index 的 access 方法，用来 从 join的 当前表中 选择行，但 where子句中的 表 可能还有 其他条件 可以 用来进一步精确 估计 传给下一个表的 合格行。

只有下面的情况，condition 才有助于 filter：

1. 它 引用的是 当前表
2. 它依赖了 join sequence 中 早期表中的 一个或多个 常量值
3. access method 没有考虑到这个 condition

在explain 的output中， rows 列 表示 所选的 access method 的行估计，filtered 列 反映 condition filtering 的效果。filtered 的值 是 百分比。 100表示 没有过滤行。 prefix row count（会从当前表 被传递到 下一个表的 row 的 number的 估计）是 rows 和 filtered 的 乘积。 也就是说， prefix row count 是 row count 的估计值，减去 filtered 估计的值。例如， 如果 rows 是 1000， filtered 是20% ， condition filtering 将 估计的 1000 减少为 1000*20%=200。

考虑下面的查询：

```
SELECT *
  FROM employee JOIN department ON employee.dept_no = department.dept_no
  WHERE employee.first_name = 'John'
  AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01';
```

假设 数据集 有以下特征：

1. employee 表有 1024 行
2. department 有 12行
3. 2个表 在 dept_no 上 都有 index
4. employee 在 first_name 上 有 index
5. 8行 满足 employee.first_name='John' 这个 condition
6. 150 行 满足 employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
7. 1行 满足 employee.first_name = 'John' AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'

没有 condition filtering， explain 的输出 如下：

```
+----+-----------+--------+-----------------+---------+---------+------+----------
-+
| id | table     | type   | possible_keys   | key     | ref     | rows | filtered
|
+----+-----------+--------+-----------------+---------+---------+------+----------
-+
| 1  | employee  | ref    | name,h_date,dept | name    | const   | 8    | 100.00
|
| 1  | department | eq_ref | PRIMARY         | PRIMARY | dept_no | 1    | 100.00
|
+----+-----------+--------+-----------------+---------+---------+------+----------
-+
```

对于 employee，对 name 这个索引 的 access method 获得了 8 条 匹配 name=John 的

row。没有 filtering (filtered shi 100)， 所以 所有 是 prefix row 的 row 都被传递到下一个表： prefix row count 就是 rows列 * filtered列 = 8 * 100% = 8。

使用 condition filtering, 优化器 还考虑了 access method 没有考虑 的 where子句中的 条件。这种情况下，优化器 使用 启发式方法 估计 对 employee.hire_date 的 between 条件的过滤效果 是 16.31%。 结果，explain 产生下面的输出：

```
+----+------------+--------+-----------------+---------+---------+------+----------
-+
| id | table      | type   | possible_keys   | key     | ref     | rows | filtered
|
+----+------------+--------+-----------------+---------+---------+------+----------
-+
| 1  | employee   | ref    | name,h_date,dept | name    | const   | 8    | 16.31
|
| 1  | department | eq_ref | PRIMARY          | PRIMARY | dept_no | 1    | 100.00
|
+----+------------+--------+-----------------+---------+---------+------+----------
-+
```

现在，prefix row count 是 rows* filtered = 8*16.31% = 1.3, 这个更接近地反映了实际数据集。
。。。但是 你 16.31 是哪里来的？ 150/1024是 14.64% . 而且 计算150 就需要 遍历 才会知道的。 根据历史query 拟合的？

通常，优化器 不会为 最后一个 join 的 表 计算 condition filtering effect(效果就是 prefix row count reduction)， 因为没有 下一个表 可以传递行。explain 出现异常：为了提供 更多信息，filtering effect 是针对 所有 joined table 计算的， 包括最后一个。

要控制 优化器 是否 考虑 additional filtering condition， 使用 optimizer_switch 系统变量 的 condition_fanout_filter 标记， 这个标志 默认启用，但是可以 禁用 来 抑制 condition filtering(例如， 你发现 特定的查询 在 没有它的情况下 性能更好。)

如果 优化器 高估了 condition filtering 的 效果， 性能可能比 不使用 条件过滤的 情况更差，，这种情况下，下面的技术 可能有所帮助：

1. 如果一个 列 没有 被index，那么 index 它 以便 优化器 获得 一些 关于 列值 分布 的信息 并且 可以 改进 它的 row 估计。
2. 同样，如果 没有 可用的 列 直方图信息，那么 生成一个 (查看 8.9.6 optimizer statistics)
3. 修改 join 顺序，实现这一点的方法 包括 join-order optimizer hints, select 后面立刻加一个 STRAIGHT_JOIN ， 和 STRAIGHT_JOIN join操作。
4. 会话级别 禁用 condition filtering

   ```
   SET optimizer_switch = 'condition_fanout_filter=off';
   ```

   或 对于 某个 query，使用 optimizer hint

   ```
   SELECT /*+ SET_VAR(optimizer_switch = 'condition_fanout_filter=off')
   */ ...
   ```

---

https://dev.mysql.com/doc/refman/8.0/en/constant-folding-optimization.html

8.2.1.14 Constant-Folding Optimization

常量和 列值 之间的比较，其中常量值 超出范围 或 相对于 列类型 的类型错误，现在 在 查询优化 期间 处理一次，而不是在 执行期间 逐行处理。 可以使用 这种方式处理的 比较是 >,>=,<,<=,<>,!=,=,<=>

考虑下面sql 创建的 表

```
CREATE TABLE t (c TINYINT UNSIGNED NOT NULL);
```

sql SELECT * FROM t WHERE c < 256 中的 where子句的 条件 包含 整数 常量 256， 超过了 tinyint unsigned 列的 范围。以前，这是通过 将 2个操作数 都视为 较大类型 来处理的，但是 现在， 由于 c 的任何允许值 都小于 常量256， 所以 where 子句 可以改写为 where 1， 这样 sql 就重写成 select * from t where 1。
。。感觉应该是 where 1=1 ，不然 后续的 ，， 或者说 where 1 and a<4 这种也是可以的？ 。 没有， where 1 不行的,至少 db2，sql server 不行。

这使得优化器 可以完全删除 where 子句， 如果 c 列可以为空 (即 c列 被定义为 tinyint unsigned) 则 查询会重写如下：

```
SELECT * FROM t WHERE ti IS NOT NULL
```

常量 和 MySQL支持的列类型 相比时， folding(折叠) 会如下 执行：

1. Integer column type。 整数类型 和 下列类型的 常量 比较时：
    1. integer value。如果 常量 超出了 类类型的 范围， comparison 被 折叠为 1 或 is not null。
    如果常量是 范围的 边界，则 比较 被折叠为 = 。例如：

```
mysql> EXPLAIN SELECT * FROM t WHERE c >= 255;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5
     filtered: 20.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `test`.`t`.`ti` AS `ti` from `test`.`t`
where (`test`.`t`.`ti` = 255)
1 row in set (0.00 sec)
```

2. floating- or fixed- point value。 如果 常量是 十进制 类型 之一 (如 decimal，real，double，float) 且 有 非0小数部分， 则它不能 equal；因此折叠(it cannot be equal; fold accordingly.)。 对于其他比较，根据 符号 向上 或 向下 舍入为 整数，然后 执行 范围检查 和 处理，就像 上面的 int-int 的处理。
   因为太小 而无法表示为 decimal 的 real 值 根据符号 四舍五入为 0.01 or -0.01， 然后作为 decimal 处理。
3. string 类型。 尝试将 string 值 解释为 integer 类型，然后 将比较 作为 2个integer值 的比较 来处理。 如果失败，则尝试 将值 作为 real 来处理。

2. decimal or real 列。decimal类型 和 下面类型的常量 的比较如下：
   1. integer值，对 列值 的 整数 部分 执行 range check。如果没有 折叠结果，则将 常量 转换为 和列值 相同小数位 的 decimal，然后 将其 作为decimal (查看下步)
   2. decimal or real 值，检查 overflow (即，常量的整数部分 是否超过了 列的 十进制类型 所允许的 位数)。 如果是的话，折叠。
      如果常量的 有效小数位数 比 列的类型 多，则 截断 常量。如果 比较运算符是 = 或 <>，则折叠。如果是 >= 或 <=，则调整operator (因为截断)。 例如，如果 列的 类型是 decimal(3,1)，则 select * from t where f>=10.13 变为 select * from t where f > 10.1
      如果常量的 小数位数 少于 列的类型，则转换为 具有相同位数的 常量。对于 real 值的 下溢(小数位数太少 而无法表示)，将常量转为 十进制的 0
   3. string值。 如果 值能被 解释为 integer 类型，那么当做 integer处理，否则，尝试作为 real 来处理。
3. float or double 列。 float(m,n) 或 double(m,n) 值 和 常量的比较 处理如下：
   1. 如果 常量 溢出了 column的range，fold
   2. 如果 常量 有多于n个小数，则截断，在折叠期间进行补偿。对于 = 和 <> 比较，折叠为 true，false，或 is [not] null。 对于其他 操作符，调整操作符
   3. 如果 常量的整数位 多余 m，则折叠。

限制。下面的情况 无法使用这项 优化：
1. 使用 between 或 in 进行 比较
2. 和 bit 列 或 使用date或time类型的 列 比较
3. 在 prepared statement 的 preparation 阶段， 尽管 这项优化 可以在 prepared statement 真正执行 的 优化阶段 应用。 但是由于 语句在 准备期间，常量的 值 还不清楚。

---

https://dev.mysql.com/doc/refman/8.0/en/is-null-optimization.html

8.2.1.15 IS NULL Optimization

MySQL 可以对 col_name is null 执行与 col_name=constant_value 相同的 优化。如，MySQL 可以使用 index 和 range 来搜索 is null。

例子：

```
SELECT * FROM tbl_name WHERE key_col IS NULL;

SELECT * FROM tbl_name WHERE key_col <=> NULL;
```

```
SELECT * FROM tbl_name
  WHERE key_col=const1 OR key_col=const2 OR key_col IS NULL;
```

如果 where子句包含了 col_name is null 条件，且 col_name 被声明为 not null，那么 这个表达式 会被 优化掉。在列 可能是 null 的情况下(如，它来自 left join 的 右表)，则不会发生这种 优化。

MySQL还 优化 col_name=expr or col_name is null 的组合， 这种形式 在 已解析 的 子查询中 很常见。使用 此优化时，explain 显示 ref_or_null。

这种优化可以 对 任何 key part 处理一个 is null

一些 被优化的 query 的例子，假设 t2 表的 a 和b列 上 有index：

```
SELECT * FROM t1 WHERE t1.a=expr OR t1.a IS NULL;

SELECT * FROM t1, t2 WHERE t1.a=t2.a OR t2.a IS NULL;

SELECT * FROM t1, t2
  WHERE (t1.a=t2.a OR t2.a IS NULL) AND t2.b=t1.b;

SELECT * FROM t1, t2
  WHERE t1.a=t2.a AND (t2.b=t1.b OR t2.b IS NULL);

SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL AND ...)
  OR (t1.a=t2.a AND t2.a IS NULL AND ...);
```

ref_or_null 首先读取 引用key，然后 单独搜索 具有 null 值的 row。

优化只能处理 一个 is null 级别。在下面query中， MySQL 只对 (t1.a=t2.a and t2.a is null) 使用 key lookup，并不能 使用 b 的 key part：

```
SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL)
  OR (t1.b=t2.b AND t2.b IS NULL);
```

---

https://dev.mysql.com/doc/refman/8.0/en/order-by-optimization.html
8.2.1.16 ORDER BY Optimization

本节描述 MySQL 何时 可以使用 index 来 满足 order by 子句，当index不能用时 使用 filesort 操作，以及 优化器 提供的 关于 order by 的 执行 计划信息。

order by 带 或 不带 limit，可能会导致 以 不同的顺序 返回 row。

Use of Indexes to Satisfy ORDER BY
一些情况下，MySQL 可能使用 index 来满足 order by 并 避免执行 filesort 操作中 涉及的额外排序。

即使 order by 和 index不完全匹配，依然可以使用index，只要 index 的 所有未使用部分 和 所有额外的 order by 中列 都是 where子句中的 常量。 如果 索引不包含 query 访问的所有 列，则仅当 index access 比 其他access方法 便宜时 才使用index。

假设 (key_part1,key_part2) 上有索引，下面的查询 可能会 使用 该index 来 解析 order by 部分。 优化器 是否真的 这样做 取决于 如果 还需要读取 不在index 中的列，那么 是通过index 还是 table scan 更有效?

1. 在这个query中，(key_part1,key_part2)上的 index 可以让 optimizer 避免 sorting:
   ```
   SELECT * FROM t1
     ORDER BY key_part1, key_part2;
   ```
   但是，query使用了 select *，这会导致 选择 不止key_part1,key_part2 的列。这种情况下，扫描整个索引 并查找表行 来找到 不在index 中的 column 可能比 扫描表+排序更昂贵。如果是这样，优化器可能不使用 index。如果select 仅选择 index列，则使用index 并 避免排序。
   如果 t1 是 InnoDB 表，表主键 是 索引的 隐式的(implicitly) 一部分， 下面的query会使用index 来解决 order by:
   ```
   SELECT pk, key_part1, key_part2 FROM t1
     ORDER BY key_part1, key_part2;
   ```

2. 在下面的query中，key_part1 是常量，所以 所有通过index 访问的 行 都按照 key_part2 的顺序，如果where子句 是 足够 selective 可以使 index range scan 比 table scan 便宜，则 (key_part1, key_part2) 上的 index 可以避免 sort。
   ```
   SELECT * FROM t1
     WHERE key_part1 = constant
     ORDER BY key_part2;
   ```

3. 下面的2个query，是否使用index， 类似于 前面的 不带 desc 的query
   ```
   SELECT * FROM t1
     ORDER BY key_part1 DESC, key_part2 DESC;

   SELECT * FROM t1
     WHERE key_part1 = constant
     ORDER BY key_part2 DESC;
   ```

4. order by 中的 2个列 可以 有相同的 方向，也可以有 不同的方向 (asc,desc)。index使用的 一个条件是 index 必须有 相同的 homogeneity(同质，同种)， 但不必具有 相同的实际方向。
   如果query 混合了 asc，desc，如果 索引也使用 相应的 混合的升序和 降序 列，则优化器可以在 列上 使用 index
   ```
   SELECT * FROM t1
     ORDER BY key_part1 DESC, key_part2 ASC;
   ```
   优化器可以使用 (key_part1,key_part2) 上的 index，如果 key_part1 是 降序的，且 key_part2 是升序的。 如果key_part1是升序，key_part2是降序， 也可以使用index，通过 backward scan(向后扫描)。

。。第一次知道 index 还有 升序，降序。。在你的索引不止一个字段，而你的查询又要对结果排序时，符合索引的排序方式会影响性能。 index只有一列时，无所谓升序降序。

5. 在下面的2个query中，将 key_part1 和一个 常量 进行比较。如果 where 子句的 selective 足够 使得 index range scan 比 table scan 更便宜，则 使用 index。

```
SELECT * FROM t1
  WHERE key_part1 > constant
  ORDER BY key_part1 ASC;

SELECT * FROM t1
  WHERE key_part1 < constant
  ORDER BY key_part1 DESC;
```

6. 下面的query中，order by 没有使用 key_part1， 但是 所有选择的 行 都有一个 常量的 key_part1 值，所以 index 依然可以使用：

```
SELECT * FROM t1
  WHERE key_part1 = constant1 AND key_part2 > constant2
  ORDER BY key_part2;
```

在某些情况下，MySQL 不能使用 index 来 解析 order by， 尽管它仍然可以 使用 index 来查找 与 where 子句匹配的行：

1. query在 不同的 index 上使用 order by

```
SELECT * FROM t1 ORDER BY key1, key2;
```

2. query 的 order by 中使用的 列 不是 index中连续的part。

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1_part1, key1_part3;
```

3. 用于获取 row 的 index 和 order by中 使用的 index 不同

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1;
```

4. query 将 order by 和 一个 包含index的 表达式 一起使用

```
SELECT * FROM t1 ORDER BY ABS(key);
SELECT * FROM t1 ORDER BY -key;
```

5. query join了许多表，且 order by中的 列 并非全部来自 用于检索row 的 第一个 非常量表 (这是 explain的输出中 第一个 没有 const join type的 表)
6. query 有不同的 order by 和 group by 表达式
7. index 是 order by 列上的 前缀索引。此时，index 不能用来 完全解析 顺序。例如，如果 仅索引 char(20) 列的 前 10个字节，则 index 无法区分 超过 10个字节的 值，因此需要 filesort 。
8. 索引不按 顺序 存储行。例如，MEMORY 表中的 HASH 索引 就不是 按顺序存储行的。

用于 sorting 的index 的可用性 可能受到 这列的别名 的使用 的影响。 假设列 t1.a 已经被 index。在这个语句中，select列表中的 列的名字 是 a ， 它引用了 t1.a， 就像 order by 中 对 a 的 引用一样， 因此可以 使用 t1.a 上的 index。

```
SELECT a FROM t1 ORDER BY a;
```

下面的语句中，select中列名也是 a， 但是它是 别名，它ref 了 abs(a)， 就像 order by 中对 a 的ref 一样，因此 不能使用 t1.a 上的 index

```
SELECT ABS(a) AS a FROM t1 ORDER BY a;
```

。。?

下面的语句中，order by 引用的 名称 不是 select中的列。但是 t1表中 有个列 名为a，所以

order by 指的是 t1.a，并且可以用 t1.a 上的 index。（当然，生成的 排序顺序 可能和 abs(a) 的顺序 完全不同）。

```
SELECT ABS(a) AS b FROM t1 ORDER BY a;
```

。。。？

以前(<= MySQL 5.7)， group by 在某些条件下 隐式排序。
在 MySQL8.0 中，这种情况不再发生，因此 不再需要 在 末尾 指定 order by null 来 抑制 隐式排序。 但是 查询的结果 可能和 以前的 MySQL 版本 不同。

Use of filesort to Satisfy ORDER BY
如果不能使用index 来满足 order by 子句，MySQL 会执行 filesort 操作，读取 table row并对它们排序。filesort 在 query 执行中 构成了一个 额外的 排序阶段。

为了 为filesort操作 获取内存，从MySQL 8.0.12 开始，优化器 会根据 需要 增量分配 内存 buffer，直到 达到 sort_buffer_size 系统变量 指定的大小，而不是 像 之前(<8.0.12)那样 预先分配 固定数量的 sort_buffer_size 个 byte。 这使得用户可以 将 sort_buffer_size 设置为 更大的值 以加快 更大的排序， 而无需担心 小的排序 会占用 过多的 内存。
(windows 上的 多个 并发 排序 可能不会 出现 这种好处，因为 它具有 弱多线程 malloc (weak multithreaded malloc))

filesort 操作 使用 临时 磁盘文件，如果 结果集 太大 而无法放入 内存。某些类型的 查询 特别 适合 在内存中的 filesort 操作。例如， 优化器 可以 使用 filesort 在 内存中 有效地处理 以下形式的 查询 (和子查询) 的 order by 操作，而 无需 临时文件。

```
SELECT ... FROM single_table ... ORDER BY non_index_column [DESC] LIMIT [M,]N;
```

此类query 在 仅显示较大结果集中的 几行的 web应用程序中 很常见

```
SELECT col1, ... FROM t1 ... ORDER BY name LIMIT 10;
SELECT col1, ... FROM t1 ... ORDER BY RAND() LIMIT 15;
```

Influencing ORDER BY Optimization
对于 没有使用到 filesort 操作的 慢速 order by 的query， 尝试 降低 max_length_for_sort_data 系统变量，降低到 适合 触发 filesort 的值。(这个 值 太高的 一个 表现 是 磁盘活动高 && cpu活动低)。 这种技术 仅 适用于 <MySQL 8.0.20， 从8.0.20 开始，max_length_for_sort_data 已经被弃用，因为 优化器做了修改，导致这个 参数无效了。
要提高 order by 速度， 请检查，是否可以让 MySQL 使用 索引 而不是 额外的排序阶段。
如果 不可能，那么 请尝试：

1. 增加 sort_buffer_size 的值，理想情况下，该值 应该足够大，以使 整个 result set 可以放入 sort buffer (避免 磁盘写入 和 merge)。
   考虑到 保存在 sort buffer 中的 列值 的 size 受到 max_sort_length 系统变量 值的影响。例如，如果 tuple 保存 长字符串的 值 且 你增加 max_sort_length 的值，那么 sort buffer tuple 的size 也会增加，你可能需要增加 sort_buffer_size。
   要监控 merge pass (to merge 临时文件) 的 数量， 检查 sort_merge_passess 状态变量
2. 增加 read_rnd_buffer_size 变量值， 以便一次读取 更多行。
3. 修改 tmpdir 系统变量 来 指向 一个 更大可用空间的 专用文件系统。 这个变量的值 可

以 list 多个路径，这些路径会被 轮询使用； 你可以使用这个 feature 来将 负载 分散到 多个 目录上。 unix上使用 : 来分隔，window 使用 ; 来分隔 路径。 路径 应该命名位于 不同 物理磁盘上的 文件系统的 目录， 而不是 同一个 磁盘上的 不同分区。

ORDER BY Execution Plan Information Available
通过 explain，你可以 检查 MySQL 是否 可以使用 index 来 解析 order by 子句：

1. 如果explain 输出 中的 extra 列 不包含 using filesort，则 使用了 index ，不执行 filesort
2. 如果 explain 输出中的extra 列 包含 using filesort，则 不使用index ，使用 filesort

另外，如果filesort 被执行了，优化器跟踪输出，包括 一个 filesort_summary 块，例如：

```
"filesort_summary": {
  "rows": 100,
  "examined_rows": 100,
  "number_of_tmp_files": 0,
  "peak_memory_used": 25192,
  "sort_mode": "<sort_key, packed_additional_fields>"
}
```

peak_memory_used 表示 排序过程中 任何一次 使用的 最大内存。 在MySQL 8.0.12之前， 输出显示 sort_buffer_size 。 (在8.0.12之前，优化器 总是 为 sort buffer 分配 sort_buffer_size 字节， 8.0.12 之后是 增量方式 为 sort-buffer 分配 byte，直到 sort_buffer_size 字节)

sort_mode 的值 提供了 有关 sort buffer 中 tuple 内容的 信息：

1. <sort_key, rowid>：这表示 sort buffer tuple 是 包含 原始 表行的 排序key值 和 row id 的 pair。元组按照 排序key 排序，row id 用于 从 表中 读取 row。
2. <sort_key, additional_fields>: 这表示 sort buffer tuple 包含 排序key值 和 查询引用的列。 元素按照 排序key 排序，列值 直接从元组中读取。
3. <sort_key, packed_additional_fields>: 和前面一样， 但 附加的列 被 紧密地 打包在一起，而不是 使用 固定长度的 编码。

explain 不区分 优化器 是否在 内存中 执行 filesort。 在 优化器 trace output 中 可以看到 内存中 filesort 的 使用。 查找 filesort_priority_queue_optimization。

---

https://dev.mysql.com/doc/refman/8.0/en/group-by-optimization.html

8.2.1.17 GROUP BY Optimization

实现 group by 子句的 最通常的方法是 扫描 整个表，并创建一个 临时表，其中每个 group 的 所有行 都是 连续的，然后 使用这个 临时表 来 发现 group 和 应用 聚合函数 。 有些情况下， MySQL 能够 做得更好，通过 index access 来避免 临时表的 创建。

为group by 使用 index 的 最重要 先决条件是 所有group by 的列 都 来自 同一个 index，且 index 按顺序 保存 键 (例如，对于 btree index 是的， 对于 hash index 不是按顺序保存键)。 零时表的使用 是否 能被index access 替代 还取决于： 查询中 使用了 index 的哪些部分， 为这部分 指定的 condition， 及 使用的聚合函数。

有2种方法 来 通过index access 执行 group by 子句。 第一种方法 将 grouping 操作 和所有 range predicates 一起应用。第二种方法 先执行 range scan，然后 对 结果 分组。

Loose Index Scan
Tight Index Scan

某些情况下，可以在没有 group by 的情况下， 使用 Loose index scan

Loose Index Scan
处理 group by 的最有效方法 是 使用 index 直接 检索 分组列。 通过这种访问方法，MySQL 使用了 一些index类型的 属性，及 key 是有序的。 这种属性 允许 在 index 中 查找 group， 而不必考虑 index 中 满足 所有 where条件的 所有 key。 这种访问方法 只考虑 index 中 一小部分 key， 因此被称为 loose index scan。
当没有where子句时，loose index scan 读取 和 group的数量 一样多的 key，这可能比 所有的key 少得多。 如果where子句包含 range predicate(参阅8.8.1 中的 range join 类型)，loose index scan 查找 满足 range condition 的 每个组的 第一个key，然后 再次读取 尽可能少的 key。在下面的condition下，是可能的：

1. query针对 单个表
2. group by 后面是 可以构成 index 最左 前缀的 列，并且不包含其他列。(如果，query 有一个 distinct 子句，而不是 group by，则所有distinct attribute 都 ref 到 构成 index 最左前缀的列)。例如，如果表t1 在 (c1,c2,c3) 上有index， 如果 query 有 group by c1,c2 ，则适用 loose index scan。 如果query 有 group by c2,c3 或 group by c1,c2,c5, 则不适用 loose index scan
3. select子句 中使用的 唯一聚合函数 是 min() 和 max()，它们都 ref到 同一列。 这列 必须在 index 中， 且 必须是 group by 中 第一个列 (。。must immediately follow the columns in the group by)
4. 除了 min(),max() 的参数外，query中的 group by 之外的 index 的 任何其他部分 都必须是 常量。 (即，它们必须 与常量 进行= 操作)
5. 对于index中的 列，必须 索引 完整的 列值，而不能是 前缀index。 前缀index 不能用于 loose index scan

如果loose index scan 适用于 query， explain 的输出中 会显示 using index for group-by ，在 extra 列。

假设在表 t1 (c1,c2,c3,c4) 上有 index idx(c1,c2,c3)。 loose index scan 可以用于 下面的query：

```
SELECT c1, c2 FROM t1 GROUP BY c1, c2;
SELECT DISTINCT c1, c2 FROM t1;
SELECT c1, MIN(c2) FROM t1 GROUP BY c1;
SELECT c1, c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT MAX(c3), MIN(c3), c1, c2 FROM t1 WHERE c2 > const GROUP BY c1, c2;
SELECT c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT c1, c2 FROM t1 WHERE c3 = const GROUP BY c1, c2;
```

下面的query 无法用 此快速select 方法 执行：

使用了 除了 min,max 外的 聚合函数：

```
SELECT c1, SUM(c2) FROM t1 GROUP BY c1;
```

group by 的列 不满足 index 的最左前缀：

```
SELECT c1, c2 FROM t1 GROUP BY c2, c3;
```

query 引用 group by 之后的 key 的 一部分， 并且不存在 与 常量 的 = 操作：

```
SELECT c1, c3 FROM t1 GROUP BY c1, c2;
```

当查询包含 where c3=const ，时， loose index scan 可以被使用。

loose index scan access method 能被 应用到 select 中其他 形式的 聚合函数，除了 min,max。

avg(distinct)，sum(distinct)，count(distinct)。avg(distinct)和sum(distinct) 接受 单个参数， count(distinct) 可以有 多个参数。

query中不能有 group by 或 distinct 子句

之前提到的 loose index scan 的 限制 依然存在。

假设，表t1(c1,c2,c3,c4) 上有 idx(c1,c2,c3)， 下面的 query 能使用 loose index scan:

```
SELECT COUNT(DISTINCT c1), SUM(DISTINCT c1) FROM t1;
SELECT COUNT(DISTINCT c1, c2), COUNT(DISTINCT c2, c1) FROM t1;
```

Tight Index Scan

tight index scan 可以是 full index scan 或 range index scan， 具体取决于 query condition。

当 不满足 loose index scan 的 条件时，依然可以 避免 为 group by 查询 创建 零时表。如果where子句中有 range condition，则该方法 只读取 满足这些 condition 的 key。否则，将执行 index scan。因为 这种方法 会 读取 where子句 定义的 每个 range 内的 所有key， 或者 如果没有 range condition 则扫描 整个index， 因此被称为 紧密index scan。使用 tight index scan，只能 在 找到 所有 满足 range condition 的 key 之后 才 执行 group 操作。

要使此方法起作用，query中所有列 有一个 恒定 的 相等 条件 就足够了， 该 条件 引用 位于 group by 的key的 之前 或 之间 的 部分key。 来自 相等 条件的 常量 填充 搜索 key 中的 任何 空白， 以便 形成 index 的完整前缀。 然后这些 index 前缀 可以用于 index lookup。

如果group by 的结果需要 排序， 并且 可以形成 作为 index 前缀的 搜索key， MySQL 也避免了 额外的 排序操作，因为有序index 中 使用 前缀 index 已经 按顺序 检索了 所有key。

假设 t1(c1,c2,c3,c4) 上 有 idx(c1,c2,c3)， 以下查询 不适用于 前面的 loose index scan， 但是 适用于 tight index scan。

group by 中有个 gap，但是 被 c2='a' 覆盖了

```
SELECT c1, c2, c3 FROM t1 WHERE c2 = 'a' GROUP BY c1, c3;
```

group by 不是 从key 的 第一个part 开始的， 但是 有一个 condition 为 该部分 提供了 一个 常量：

```
SELECT c1, c2, c3 FROM t1 WHERE c1 = 'a' GROUP BY c2, c3;
```

---
https://dev.mysql.com/doc/refman/8.0/en/distinct-optimization.html
8.2.1.18 DISTINCT Optimization

distinct 和 order by 一起使用， 在很多情况下 需要 一个 临时表。

因为distinct 可能使用 group by， 所以了解 MySQL 如何处理 order by 或 having 子句中 不属于 select 的列。

大多数情况下，可以将 distinct 视为 group by 的特例。 例如，下面的 2个 查询 是等价的。

```
SELECT DISTINCT c1, c2, c3 FROM t1
WHERE c1 > const;

SELECT c1, c2, c3 FROM t1
WHERE c1 > const GROUP BY c1, c2, c3;
```

由于这种 等价性， 适用于 group by 的优化 也可以应用于 等价的 distinct 。因此， 可以看 上一节 的 group by optimization

当 limit row_count 和 distinct 结合时， MySQL 在 搜索到 row_count 个 唯一 row 时停止。

如果 你没有用到 query中声明的 所有表 中的 列， MySQL 会在找到 第一个匹配项 后 立刻停止 扫描 任何 没有使用的 表。
下面，假设 t1 在 t2 之前 使用 (你可以使用 explain 检查)， 当 MySQL 在 t2 中找到 第一行时，它会 停止 从 t2 读取 (对于 t1 中的 任何 特定行)：

```
SELECT DISTINCT t1.a FROM t1, t2 where t1.a=t2.a;
```

---
https://dev.mysql.com/doc/refman/8.0/en/limit-optimization.html

8.2.1.19 LIMIT Query Optimization

如果你只需要 result set中 指定数量的 行，请在查询中使用 limit 子句。

MySQL 有时会优化 具有 limit row_count 子句 且 没有having 子句的 查询：

1. 如果 limit 只选择了 a few row， MySQL 在某些情况下 会使用index，通常 它更愿意 full table scan
2. 如果 limit row_count 和 order by 联用， MySQL 会在 找到 排序的结果的 前

row_count 行 后 停止，而不是 对 整个result 进行 排序。 如果 通过index 进行排序，速度很快。 如果 必须进行 filesort， 所有匹配的(不带limit子句的) 行 被 选中，对 它们中的 大部分 或 全部 进行 排序。 在找到 row_count 行后，MySQL 不会 结果集 的剩余 部分进行 排序。
这种行为的 一种表现是， 带有 和 不带有 limit 的 order by 查询 可能会以 不同的顺序 返回行。

3. 如果将 limit 和 distinct 结合使用，mysql 会在找到 row_count 唯一行 后 停止。
4. 在某些情况下，可以通过 按顺序读取 index (或对index 进行sort) 来解决 group by，然后计算 摘要 直到 index 值变化。在这种情况下，limit 不会计算 任何不必要的 group by 值。
5. 一旦MySQL向客户端 发送了所需数量的row， 它就中止查询，除非你使用 SQL_CALC_FOUND_ROWS，在这种情况下，可以使用 select found_rows() 来检索 行数。
6. limit 0 快速返回一个 空集。这对于检查 query的 有效性 很有用。它还可以用于 获取使用 MySQL API 的 应用程序 中 结果列的 类型，该API 使 结果集 元数据 可用。使用 mysql 客户端程序，你可以使用 --column-type-info 选项 来显示 结果列 类型。
7. 如果 服务器使用 临时表来解析 query，它使用 limit row_count 子句 来 计算 需要多少空间。
8. 如果index 不用于 order by 但也存在 limit 子句，则优化器 可能能够 避免 使用 merge file并 在内存中 对 row 进行 sort。

如果多行 在 order by 列中 具有相同的值，则服务器可以自由滴以任何顺序 返回这些行，并且可能会根据 整体执行计划 以不同的方式 返回。 换句话说，这些行的排序顺序对于 无序的列 是不确认的。

影响执行计划的 一个 因素 是limit， 因此，带和不带 limit 的 order by 查询 可能会以不同的 顺序 返回 row。 考虑下面的查询，它按 category 列 进行排序， 但是 id 和 rating 列 是不确定的：

```
mysql> SELECT * FROM ratings ORDER BY category;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  3 |        2 |    3.7 |
|  4 |        2 |    3.5 |
|  6 |        2 |    3.5 |
|  2 |        3 |    5.0 |
|  7 |        3 |    2.7 |
+----+----------+--------+
```

包括limit 可能会影响 每个 category 值 中的 行顺序。例如，下面是 一个 有效的 查询结果：

```
mysql> SELECT * FROM ratings ORDER BY category LIMIT 5;
+----+----------+--------+
| id | category | rating |
+----+----------+--------+
|  1 |        1 |    4.5 |
|  5 |        1 |    3.2 |
|  4 |        2 |    3.5 |
```

```
|  3 |        2 |    3.7 |
|  6 |        2 |    3.5 |
+----+----------+--------+
```

。。 但是 应该是 幂等的吧？ 应该有一个 隐式顺序吧。 肯定的， 不然分页爆炸。。

在所有情况下，row 都按照 order by 的列 排序，这是 标准SQL 要求的。

如果要确保 使用和不使用limit 的 行顺序 相同， 请在 order by 中 包含其他列 以使得 顺序具有 确定性。 例如，包含 id 。

对于带有 order by 或 group by 和 limit 子句的 查询， 优化器 默认情况下 会 尝试 选择有序index，这样可以加快 执行速度。 在 MySQL8.0.21 之前， 无法 覆盖这种行为，从 8.0.21 开始，可以通过 将 optimizer_switch 系统变量的 prefer_ordering_index 标志设置为 off 来关闭 此优化。

实例： 首先 我们 创建 和 填充 一个表 t：

```
# Create and populate a table t:

mysql> CREATE TABLE t (
    ->     id1 BIGINT NOT NULL,
    ->     id2 BIGINT NOT NULL,
    ->     c1 VARCHAR(50) NOT NULL,
    ->     c2 VARCHAR(50) NOT NULL,
    ->  PRIMARY KEY (id1),
    ->  INDEX i (id2, c1)
    -> );

# [Insert some rows into table t - not shown]
```

查看 prefer_ordering_index 是否启用：

```
mysql> SELECT @@optimizer_switch LIKE '%prefer_ordering_index=on%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%prefer_ordering_index=on%' |
+-----------------------------------------------------+
|                                                   1 |
+-----------------------------------------------------+
```

由于 下面的 query 有一个 limit 子句，我们希望 它尽可能 使用 有序index。 在这种情况下，正如 我们从 explain 的输出中看到的， 它使用 表的 主键。

```
mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: index
```

```
possible_keys: i
          key: PRIMARY
      key_len: 8
          ref: NULL
         rows: 2
     filtered: 70.00
        Extra: Using where
```

禁用 prefer_ordering_index， 重新执行上面的 query， 这次，它使用了 索引 i (这个锁定 包含了 where中使用的 c2) 和 filesort

```
mysql> SET optimizer_switch = "prefer_ordering_index=off";

mysql> EXPLAIN SELECT c2 FROM t
    ->     WHERE id2 > 3
    ->     ORDER BY id1 ASC LIMIT 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 8
          ref: NULL
         rows: 14
     filtered: 100.00
        Extra: Using index condition; Using filesort
```

---

https://dev.mysql.com/doc/refman/8.0/en/function-optimization.html

8.2.1.20 Function Call Optimization

MySQL 的函数 在内部 被标记为 deterministic 或 nondeterministic (确定性，非确定性)。一个函数 是不确定的，如果给它的参数 是固定值，它可以为 不同的 调用返回不同的 结果。非确定性函数的 例子是 rand(),uuid()。

如果一个函数 被标记为 非确定性的，则在 where 子句中 对 每行row 进行 其值的评估。

MySQL 还根据 参数类型，参数是列还是 常量 决定何时 eval 函数。 当 列值 被修改时， 对这个列 进行的 确定性函数 需要重新eval

非确定性函数 可能会 影响 查询性能。例如，某些优化 可能不可用，或者 可能需要更多的 lock。 下面讨论 rand()， 但是 这些讨论也适用于 其他 非确定性函数

假设表有下列定义

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY, col_a VARCHAR(100));
```

考虑下面的2个查询：

```
SELECT * FROM t WHERE id = POW(1,2);
SELECT * FROM t WHERE id = FLOOR(1 + RAND() * 49);
```

2个查询 似乎都使用了 主键 lookup，因为 id = xx 这个条件，但是 实际上 只有第一个使用了 主键lookup：

第一个查询 总是 最多产生 一行，因为 带有 常量参数的 pow() 是一个常量值，用于 index lookup

第二个query 包含了 一个使用 非确定性函数 rand 的表达式，该函数 在查询中不是常量，每行都是一个 新值。因此，query 读取 表的 每行，并对 每行 eval predicate，并输出 主键 与 随机值 匹配的 所有行。这可能是 0,1，或多行。

不确定性的 影响不仅限于 select 语句。 这个update 语句 使用了 非确定性 函数 来选择要修改的 row：

```
UPDATE t SET col_a = some_expr WHERE id = FLOOR(1 + RAND() * 49);
```

刚才描述的行为，对 性能和复制的 影响：

1. 由于 非确定性函数 不会 产生 常量值，优化器 无法使用 一些策略，如 index lookup。最终可能是 table scan
2. InnoDB 可能会 升级为 range-key lock ，而不是 为匹配的 row 获得 行锁。
3. 使用 不确定性函数的 update 对于 复制来说 是不安全的。

困难源于这样一个事实：rand 函数 需要 对表的 每行都eval， 为了避免 多次 函数eval，请使用下面的计数：

1. 将包含 不确定性函数的 表达式 移至 单独的语句，将值保存在 变量中。 在原始语句中，将表达式 替换为 对 变量的 ref， 优化器可以将其视为常量值：

```
SET @keyval = FLOOR(1 + RAND() * 49);
UPDATE t SET col_a = some_expr WHERE id = @keyval;
```

2. 将随机值分配给 派生表 的变量。这种技术导致 变量在 where子句中的 比较 的使用之前被 赋值一次。

```
UPDATE /*+ NO_MERGE(dt) */ t, (SELECT FLOOR(1 + RAND() * 49) AS r) AS dt
SET col_a = some_expr WHERE id = dt.r;
```

如前所述，where子句中的 非确定性 表达式 可能会阻止 优化 并导致 全表扫描。但是，如果其他表达式 是确定的，则可以 部分优化 where子句：

```
SELECT * FROM t WHERE partial_key=5 AND some_column=RAND();
```

如果优化器可以使用 partial_key 来减少 所选行的 集合，则rand()的执行次数会减少，从而减少 非确定性对 优化的影响。

---

https://dev.mysql.com/doc/refman/8.0/en/window-function-optimization.html

8.2.1.21 Window Function Optimization

。。what is window function's chinese name ?

window function 会影响优化器考虑的策略：

1. 如果subquery 有 window function ，则 禁用 子查询的 派生表merging。子查询总是 materialized(物化的，成为现实，发生)
2. semijoin 不适用于 window function optimization，因为 semijoin 适用于 where 和 join .. on 的子查询，不能包含 window function
3. 优化器 按顺序 处理 具有 相同排序要求的 多个window，因此 对于 第一个之后的窗口 可以跳过 排序。
4. 优化器不会尝试 合并 可以在 单个步骤中评估的 window (例如，当多个 over子句 包含 相同的 window 定义)。 解决方法是在 window子句中定义 window 并在 over子句中 ref window 名字。

没有用作 window function 的 聚合函数 在 最可能的 外层的 query中 聚合。 例如，在下面的query中，MySQL 发现 count(t1.b) 是外部 查询中不存在的东西，因为它位于 where子句中：

```
SELECT * FROM t1 WHERE t1.a = (SELECT COUNT(t1.b) FROM t2);
```

因此，MySQL 在子查询中 进行聚合，将 t1.b 视为常量 并返回 t2的 行数。
。。?

将where替换为 having，会导致error

```
mysql> SELECT * FROM t1 HAVING t1.a = (SELECT COUNT(t1.b) FROM t2);
ERROR 1140 (42000): In aggregated query without GROUP BY, expression #1
of SELECT list contains nonaggregated column 'test.t1.a'; this is
incompatible with sql_mode=only_full_group_by
```

发生错误是因为 count(t1.b) 可以存在 having 中，因此使用了 外部查询聚合。

window function (包括 用作 window function 的 聚合函数) 没有上述 复杂性。它们总是在编写它们的 子查询中聚合， 而不是在 外部查询中。

window function evaluation 可能收到 windowing_use_high_precision 系统变量的 影响，这个变量 决定了 是否在 不损失精度 的情况下 计算 window 操作。 默认下，这个变量是 启用的。

对于某些 moving frame aggregates， 可以应用 inverse aggregate function（逆聚合函数）从 聚合中 删除值。 这可以 提高性能，但可能会 降低 精度。 例如，将非常小的 浮点值 添加到 非常大的 值，会导致 小值 被 大值 "隐藏"， 稍后的 反转大值 会导致 小值的效果 消失。

由于反向聚合导致的 精度损失 仅是 浮点(近似值) 数据类型 操作的 因素。 对于其他类型，反向聚合 是安全的，比如 decimal，它允许小数部分，但是它是精确值。

为了更快执行，MySQL 总是在安全的情况下使用 反向聚合：

1. 对于浮点值，反向聚合并不总是安全的， 可能会导致 精度损失。 默认是 避免 反向聚合，它速度较慢，但 保持精度。如果 允许为了 速度 而牺牲 安全性，则可以禁用

windowing_use_high_precision 以允许 反向聚合。

2. 对于 非浮点数据类型， 反向聚合 始终是安全的，并且无论 windowing_use_high_precision 是什么，都可以使用。
3. windowing_use_high_precision 对于 min 和 max 没有影响，它们在 任何情况下 都不使用 反向聚合。

对于 方差函数(variance function) stddev_pop,stddev_samp,var_pop,var_samp，和它们的同义词， eval 可以在 优化模式 或 默认模式 下进行。 优化模式 可能会在 最后一个 有效数字 产生 略有不同的 结果。 如果 允许此类差异， 那么 禁用 windowing_use_high_precision 以允许 优化模式。

对于explain， windowing execution plain 信息 过于 广泛，无法以传统的 输出 格式 显示。要查看 windowing information，使用 explain format=json， 并查找 windowing 元素。

---

https://dev.mysql.com/doc/refman/8.0/en/row-constructor-optimization.html

8.2.1.22 Row Constructor Expression Optimization

row constructor 允许 同时比较多个值，例如，下面2个语句 在 语义上 是相等的：

```
SELECT * FROM t1 WHERE (column1,column2) = (1,1);
SELECT * FROM t1 WHERE column1 = 1 AND column2 = 1;
```

此外，优化器 以相同的方式 处理 这2个表达式

如果 row constructor column 不覆盖 index 的前缀，则 优化器 不太可能 使用 index。考虑下表， 它在 (c1,c2,c3) 上有一个主键：

```
CREATE TABLE t1 (
  c1 INT, c2 INT, c3 INT, c4 CHAR(100),
  PRIMARY KEY(c1,c2,c3)
);
```

在这个query中， where子句使用 index 的所有列， 但是，row constructor 本身没有 覆盖索引前缀，所以 优化器 只使用了 c1 (key_len=4，the size of c1)：

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1=1 AND (c2,c3) > (1,1)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 3
     filtered: 100.00
```

```
           Extra: Using where
```

。。？是指 select * 是 4列， 它不是前缀，导致 只是用了 c1 ？ 但是？？？
。。还是说 row constructor 是指 (c2,c3)>(1,2) ？？？ 结合下面 应该是的。。。

这种情况下， 使用 等效的 非constructor 表达式 重写 row constrcutor 表达式 可能会导致 更完整的 index 使用。 对于给定的查询， row constructor 和 等效的 非constructor 表达式 是：

```
(c2,c3) > (1,1)
c2 > 1 OR ((c2 = 1) AND (c3 > 1))
```

。。感觉是 字典顺序，， 因为 (2,0) 也是符合的 。 或者 就是它等效的那个。

重写后，优化器使用了 index 的3列 (key_len=12)

```
mysql> EXPLAIN SELECT * FROM t1
       WHERE c1 = 1 AND (c2 > 1 OR ((c2 = 1) AND (c3 > 1)))\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 12
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using where
```

因此，为了更好的结果，避免将 row constructor 和 and/or表达式 混用。 只是用一种。

某些情况下，优化器可以将 range access method 应用到 具有 row constructor 参数的in()表达式

---

https://dev.mysql.com/doc/refman/8.0/en/table-scan-avoidance.html

8.2.1.23 Avoiding Full Table Scans

当mysql 使用 full table scan 来解析query时， explain 在 type列 显示 ALL。这通常发生在 以下情况：

1. table 很小，以至于 table scan 比 key lookup 快。 这对于 少于 10行 且 行长较短的表 很常见。
2. index 的列 的 on 或where 子句中 没有 可用的 限制。
3. 你正在 将 index列 和 常量值 进行比较，并且 mysql(基于index tree) 已经计算 出：

常量覆盖了 表的 大部分，所以 table scan 更快。

4. 你正在通过 另一列 使用 具有 low cardinality 的key (许多行 和 key值 匹配)。 在这种情况下， mysql 假设通过 使用 key 可能需要 更多次 key lookup， table scan 可能更快。

对于小表， table scan 通常是合适的， 性能影响可以 忽略不计。 对于 大表，请尝试 下面的技术来 避免 优化器 错误地 选择 table scan:

1. 使用 analyze table tbl_name 来 更新 表的 key distribution。
2. 使用 force index 来告诉 mysql 表扫描 比 使用给定的index 更贵：

```
SELECT * FROM t1, t2 FORCE INDEX (index_for_column)
  WHERE t1.col_name=t2.col_name;
```

3. 启动mysqld 时 使用 --max-seeks-for-key=1000 选项，或使用 set max_seeks_for_key=1000 来告诉 优化器： 假设没有 key scan 会 导致 超过 1000次的 key seek。

==========

https://dev.mysql.com/doc/refman/8.0/en/subquery-optimization.html

8.2.2 Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions

MySQL query 优化器 有不同的策略 用于 subquery 的 eval：

对于 使用了 in,= ANY,或 exist 的 子查询，优化器有以下 选择：

semijoin,materialization,exists strategy

对于 使用 not in, <> ALL， not exists 的 子查询，有以下选择：

materialization,exists strategy

对于派生表，优化器有以下选择(也适用于 view reference 和 common table expression):

merge 派生表 到外部 查询块

将派生表 materialize 为 内部 临时表

对于使用 子查询来修改 单个表 的 update 和 delete 语句 的 限制 是 优化器 不使用 semijoin 或 materialization 子查询优化。 作为一种 解决方法，尝试将 它们重写为 使用 join 而不是 子查询的 多表 update 和 delete语句。

---

https://dev.mysql.com/doc/refman/8.0/en/semijoins.html

8.2.2.1 Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations

。。SEMI join 简单来说就是 外层的表的过滤依赖于内层的子查询语句作为过滤条件。

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html

Chapter 15 The InnoDB Storage Engine

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html

15.1 Introduction to InnoDB

是一个兼顾 高可靠性 和 高性能 的 通用存储引擎.
MySQL 8.0 是 默认的 存储引擎。 不带engine子句的 create table 会 创建 InnoDB 表。

InnoDB主要优势

1. 它的DML操作 遵循 ACID 模型，具有 commit,rollback,crash-revocery(崩溃恢复) 功能的 事务 以保护用户数据。
2. row-level locking 和 oracle风格的一致性读取 提高了 多用户 并发 和 性能。
3. InnoDB 表 在 磁盘上 arrange 数据 以 优化 基于 主键的 query。每个InnoDB表 都有 一个 称为 聚簇索引的 主键索引，它组织数据 以最小化 主键查找的 IO。
4. 为了保证数据完整性，InnoDB支持 foreign key 约束。使用了外键的 insert,update,delete 会确保 它们不会导致 相关表 之间的 不一致。

Table 15.1 InnoDB Storage Engine Features

Feature	Support
B-tree indexes	Yes
Backup/point-in-time recovery (Implemented in the server, rather than in the storage engine.)	Yes
Cluster database support	No
Clustered indexes	Yes
Compressed data	Yes
Data caches	Yes
Encrypted data	Yes (Implemented in the server via encryption functions; In MySQL 5.7 and later, data-at-rest encryption is supported.)

Foreign key support	Yes
Full-text search indexes	Yes (Support for FULLTEXT indexes is available in MySQL 5.6 and later.)
Geospatial data type support	Yes
Geospatial indexing support	Yes (Support for geospatial indexing is available in MySQL 5.7 and later.)
Hash indexes	No (InnoDB utilizes hash indexes internally for its Adaptive Hash Index feature.)
Index caches	Yes
Locking granularity	Row
MVCC	Yes
Replication support (Implemented in the server, rather than in the storage engine.)	Yes
Storage limits	64TB
T-tree indexes	No
Transactions	Yes
Update statistics for data dictionary	Yes

To compare the features of InnoDB with other storage engines provided with MySQL, see the Storage Engine Features table in Chapter 16, Alternative Storage Engines.

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-benefits.html

15.1.1 Benefits of Using InnoDB Tables

InnoDB有下面的好处：

1. 如果server 因为 软件或硬件问题 而意外退出，无论当时数据库中发生了什么，重启数据库后 都不需要做任何特别的事情。InnoDB crash recovery 自动完成 崩溃前提交的 更改，并撤销 正在处理但没有commit的更改，从而允许你 重新启动 并从 中断处 继续。
2. InnoDB存储引擎 维护自己的 buffer pool，在访问 数据时 将表 和 index 数据 缓存在内存中。经常使用的数据 直接从 内存中处理。这个cache 适用于 多种类型的 信息 并加快处理速度。 在专用的 数据库server上， 多达80% 的物理内存 通常 分配给 buffer pool。
3. 如果将相关数据拆分到 不同的表中，则可以设置 外键 来强制 引用完整性。
4. 如果磁盘或内存中的 数据损坏，checksum 机制会 在你使用 虚假数据前 向你发出警告。innodb_checksum_algorithm 变量定义了 InnoDB 使用的 checksum 算法。
5. 当你为设计了一个 每个表都具有 适当主键列的 数据库时，涉及这些列的操作 会自动优化。 在 where子句，order by 子句，group by子句 和 join操作中 引用 主键列 非常快。
6. insert,update,delete 被自动优化，这种机制叫 change buffering。InnoDB 不仅允许 对同一张表 进行 并发读写，它还缓存 更改的数据 以 streamline(更高效，节约) 磁盘IO。

7. 性能优势 不仅仅 限于 那些 会长时间运行 query 的 大型表。当从 表中 反复访问 相同的行时， 自适应hash index 会接管 以使 这些query 更快，就好像 它们来自 hash表一样。
8. 你可以压缩 表 和 关联的index
9. 你可以加密你的数据
10. 你可以创建和删除 index 并执行 其他ddl操作，而对性能和可用性 的影响要小很多。
11. 截断(truncating) 一个 file-per-table 的 tablespace 非常快，并且可以释放 磁盘空间让OS使用。
12. 对于blob 和 长文本 字段，表数据的 存储布局 采用了dynamic row format， 更有效。
13. 你可以通过 查询 information_schema 表 来 监控 存储引擎的 内部工作。
14. 你可以通过 查询 performance schema 表 来监控 存储引擎的 性能细节。
15. 你可以将 InnoDB 表 和 其他存储引擎的表 混合，甚至在同一语句中。例如，你可以使用 join 操作 将 InnoDB 和 MEMORY 表中的 数据 组合到 单个query中。
16. InnoDB 专为 处理 大数据量时 的 CPU效率 和 最大性能 而设计。
17. InnoDB表可以处理 大量数据，即使在 文件大小限制为 2g 的OS上也是如此。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-best-practices.html

15.1.2 Best Practices for InnoDB Tables

本节介绍使用 InnoDB 表时的 最佳实践：

1. 为每个表 指定主键， 主键选择 那些 被 最频繁查询的 那些列， 如果没有明显的主键，就指定一个 auto-increment value。
2. 使用 join 在那些 基于表中相同的id值 从多个表中 提取数据的 地方。 为了 join性能，在 join column 上定义 foreign key，并 在每个表中 声明 这些列 具有 相同的 数据类型。添加外键 可以确保 对引用 的 列 进行 index，从而提高性能。外键 还将 删除 和 update 传播到 所有 受影响的 表。 如果 父表中 不存在 相应的id，则防止在 子表中 插入数据。
3. 关闭自动commit。每秒上百次的commit 会 影响性能（受存储设备的 写入速度限制）
4. 通过使用 start transaction 和 commit 来将 相关的 DML 操作组合成事务。
5. 不要使用 lock tables。 InnoDB可以处理 多个 对相同表的 读写 会话，而不会 牺牲 可靠性 和 高性能。 要使用 对一组行的 独占写入访问权限， 请使用 select .. for update 来锁定 你准备 update 的 row
6. 启用 innodb_file_per_table 变量 或 使用 通用tablespace 来将 表的数据 和index 放入 单独的 文件 而不是 system tablespace。 innodb_file_per_table 是默认启用的。
7. 评估 你的数据 和访问 模式 是否 受益于 InnoDB表或page 压缩功能。你可以在 不牺牲读写能力的 情况下 压缩 InnoDB表。
8. 启动 server是 使用 --sql_mode=NO_ENGINE_SUBSTITUTION 选项， 以防止 使用 你不想使用的 存储引擎 创建表。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-check-availability.html

15.1.3 Verifying that InnoDB is the Default Storage Engine

使用 show engines 来查看 可用的 存储引擎， 在 support列中寻找default
```
mysql> SHOW ENGINES;
```

或者查询 INFORMATION_SCHEMA.ENGINES 表
```
mysql> SELECT * FROM INFORMATION_SCHEMA.ENGINES;
```

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-benchmarking.html

15.1.4 Testing and Benchmarking with InnoDB

如果InnoDB 不是默认存储引擎，你可以通过在 命令行中定义 --default-storage-engine=InnoDB 或 在mysql服务器 option file 的 [mysqld]部分 使用 default-storage-engine=innodb。

由于更改默认存储引擎 只会影响 新建的表，所以 运行应用 来确保 正常。如果表依赖于 特定于另一个 存储引擎的 功能， 你会 收到 错误信息。在 这种情况下， 将 engine=other_engine_name 子句 添加到 create table 语句 以避免错误。

如果你没有对 存储引擎做出深思熟虑的决定，并且 想要预览 使用 InnoDB 创建的表 的 工作情况， 可以使用 alter table table_name engine=InnoDB 修改表的 引擎。 或者，要在 不影响原始表的情况下 测试 查询和其他语句，请创建副本：
```
CREATE TABLE ... ENGINE=InnoDB AS SELECT * FROM other_engine_table;
```

要在 实际工作负载下 评估完整 应用程序的 性能，请安装 最新的 mysql 服务器 并运行 基准测试 (benchmarks)

测试整个应用程序生命周期，从安装到 大量使用，再到 服务器重启。 在数据库 繁忙时kill 掉，并在 重启时 验证 数据是否 恢复成功。

---
https://dev.mysql.com/doc/refman/8.0/en/mysql-acid.html

15.2 InnoDB and the ACID Model

ACID 模型是一组数据库设计原则，强调 对业务数据 和 关键任务应用 很重要的 可靠性。
MySQL 的引擎 是ACID 的。
如果你有额外的 软件保护措施，可靠的硬件 或 可以容忍少量数据丢失 或不一致的 应用，你可以 调整 MySQL 设置 以交换 一些 ACID 可靠性 来 获得 更高的 性能 或 吞吐量

下面讨论MySQL的特性，特别是InnoDB 存储引擎， 如何 与ACID 模型 进行 交互。

Atomicity
主要涉及InnoDB的事务，相关的 mysql 功能是：
- autocommit
- commit
- rollback

Consistency
主要涉及内部 InnoDB处理 以保护 数据 免受崩溃，相关mysql功能：
- InnoDB doublewrite buffer
- InnoDB crash recovery

Isolation
主要涉及 InnoDB的事务，特别是 适合于 每个事务的 隔离级别 ，相关mysql 功能：
- autocommit
- 事务隔离级别 和 set transaction 语句。
- InnoDB locking 的 底层细节。 可以在 information_schema 表 和 Performance schema data_locks，data_lock_waits 表 中查看 详细信息

Duarbility
相关mysql功能：
- InnoDB doublewrite buffer
- innodb_flush_log_at_trx_commit 变量
- sync_binlog 变量
- innodb_file_per_table 变量
- 存储设备中的 write buffer，例如 磁盘驱动，SSD，RAID
- 存储设备的 battery-backed cache
- 运行mysql的 OS，特别是 它对 fsync() system call 的支持。
- UPS(uninterruptible power supply) 保护 运行mysql server 和 存储 mysql 数据的 所有 服务器和存储设备的 电力
- 你的备份策略，如 备份频率 和 类型，以及 备份保留期
- 对于 分布式 或 hosted 数据应用程序， MySQL 服务器 硬件 所在 的数据中心的 特定特征，以及 数据中心之间的 网络连接。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html

15.3 InnoDB Multi-Versioning

InnoDB 是一个 多版本存储引擎。 它保留有关 已更改行的 旧版本的信息，以支持 并发 和 回滚 等事务功能。这些信息 保存在 被称为 rollback segment 的 数据结构中 的 undo tablespace。
InnoDB使用 rollback segment 中的信息 来执行 事务 回滚 所需的 撤销操作。 它还使用这些信息 来 构建row 的早期版本 以进行一致性读取。

在内部，InnoDB为存储 在数据库中 为每一行 增加 3个字段：

1. 一个6byte 的 DB_TRX_ID，指示 inserted 或 updated row 的 最后一个 事务的 事务标识符。 此外，删除在 内部 被视为 update， 其中 行中的 特殊位 设置为 该行已删除。
2. 7byte 的 DB_ROLL_PTR，称为 roll pointer。 roll pointer 指向 写到 rollback segment 的 undo log record。 如果行已更新，则 undo log record 包含 用于重建update前的 行内容 所需的 信息。
3. 6byte的 DB_ROW_ID，包含一个 row id，这个row id 随着 新行的insert 而 单调递增。 如果 InnoDB自动生成 聚集索引，则索引包含 row id 值。否则，DB_ROW_ID 不会出现在任何index 中。

rollback segment中的 undo log 分为 insert undo log，update undo log。 仅在事务回滚时才需要 insert undo log，并且可以在 事务提交后 立即丢弃。 update undo log 也用于 一致性读取，但是 只有在 不存在 InnoDB 为其分配快照的 事务后 才能丢弃它们，在一致性读取中 可能需要 update undo log 的信息 来 构建 早期版本的 数据库行。

建议你 定期提交事务，包括 仅发出一致读取的事务。 否则，InnoDB 无法丢弃 update undo log中的数据，并且 rollback segment 可能会变得太大，填满 它所在的 undo tablespace。

rollback segment 中 undo log record 的物理size 通常 小于 相应的 inserted 或 updated row。你可以使用 此类信息 来计算 rollback segement 所需的 空间。

在 InnoDB multi-versioning schema中， 当你使用 sql 删除 row 时，不会立即 从数据库中物理删除它。 InnoDB 仅在丢弃 为delete而写入的 update undo log record 时 才 物理删除相应的row 和 index。 这种删除操作被称为 purge， 它非常快，通常与 执行删除的 sql语句所用的 时间顺序 相同。

如果你在 表中 以 大致相同的 速度 以 小批量insert 和 delete row，则 purge 线程 可能会滞后，并且 由于 "dead" row，表会越来越大，会受到磁盘空间 影响，并且 性能降低。 这种情况下，通过 调整 innodb_max_purge_lag 系统变量 来 限制 new row operation，并为 purge 线程 分配 更多资源。

Multi-Versioning and Secondary Indexes
InnoDB multiversion concurrency control (MVCC) 对二级索引 的处理 和 聚簇索引 不同。聚簇索引 中的记录在 原地更新，它们的 隐藏的系统列 指向了 undo log 。 和聚簇索引不同， 二级索引记录 不包含 隐藏的系统列，也不会 原地更新。

当二级索引列 被更新时，旧二级索引记录 被标记为 delete，新的记录insert进来，被标记为 delete 的记录 最终会被 purge。 当二级索引记录 被 标记为删除 或二级索引page 被 更新的事务 更新时，InnoDB在聚集索引中 查找数据库记录。 在聚集索引中，检查记录的 DB_TRX_ID，如果在启动读取事务后 修改了记录，则从 undo log 中检索 记录的正确版本。

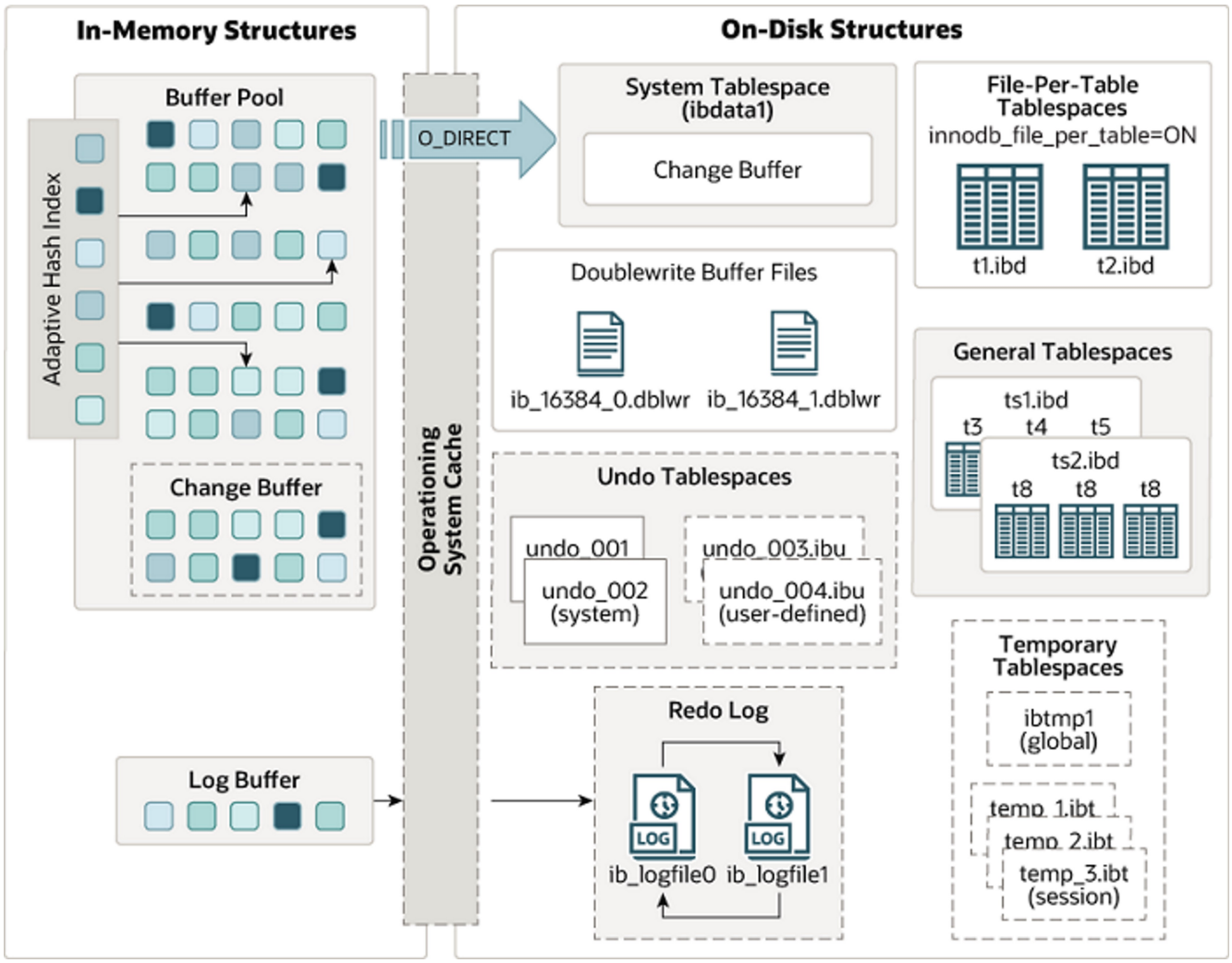
如果二级索引记录 被标记为 删除 或 二级索引page 被 更新的事务更新， 则不使用 covering index 技术。 InnoDB从聚集index中 查找记录 而不是从 索引结构中。

但是，如果启用了 index condition pushdow(ICP) 优化，并且可以 仅使用 index 的字段 来评估 where 条件的一部分，MySQL服务器 仍然会将这部分 where条件 下推到 评估它的 存储引擎，存储引擎会使用 index 来eval。如果没有找到匹配的记录，则避免聚集索引查找。如果找到匹配的记录，即使记录被标记为delete，InnoDB 也会在 聚集索引中 查找记录。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-architecture.html
15.4 InnoDB Architecture

下面的图 显示了 组成 InnoDB 存储引擎架构 的 内存和磁盘结构。



---
https://dev.mysql.com/doc/refman/8.0/en/innodb-in-memory-structures.html

15.5 InnoDB In-Memory Structures

15.5.1 Buffer Pool
15.5.2 Change Buffer
15.5.3 Adaptive Hash Index
15.5.4 Log Buffer

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html

15.5.1 Buffer Pool

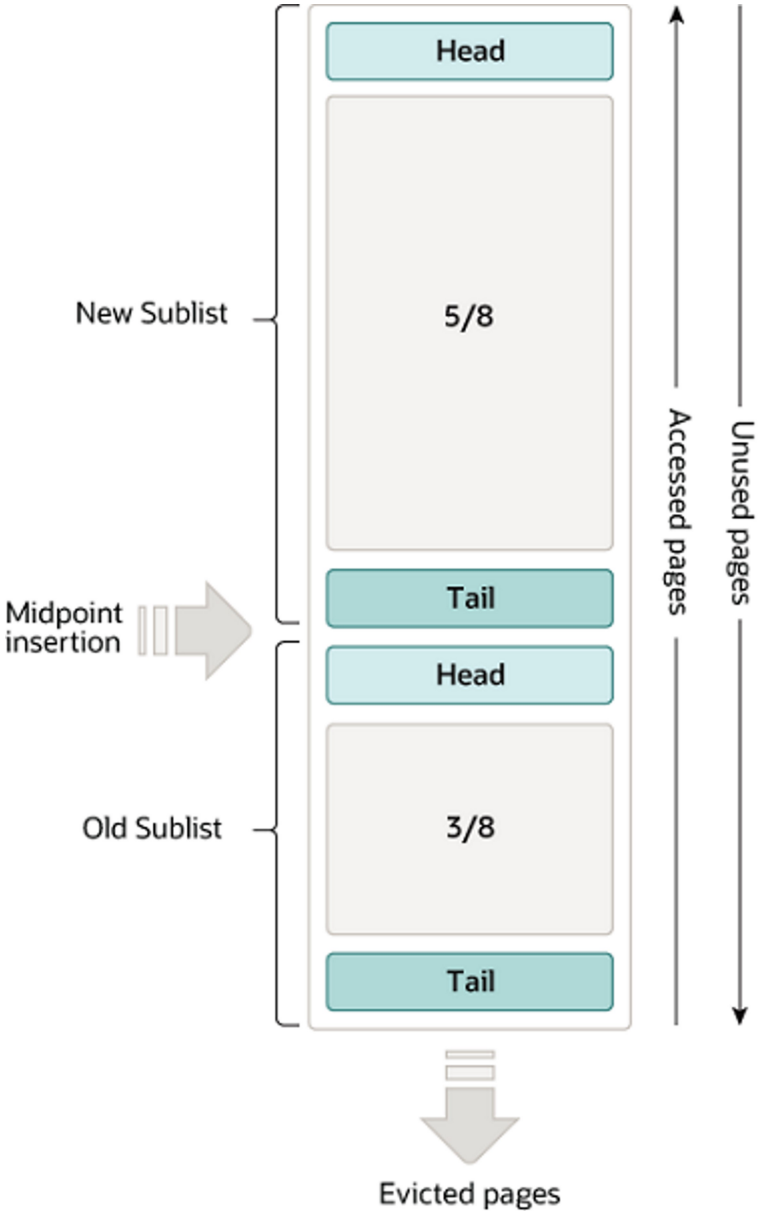
buffer pool 是内存中的 一个区域， InnoDB在访问时 cache 表和index 的数据。buffer poll 可以让 经常使用的数据 直接从内存中获得， 从而加快速度。在 专用server上，多打80%的物理内存通常分配给 buffer pool。

为了提高 大容量读取 操作的效率， buffer pool 被划分为 可能包含多行 的 page。为了有效地管理缓存，buffer pool 被实现了 a linked list of pages。 使用LRU的变体 来移除数据。

了解如何利用buffer pool 将 频繁访问的数据 保存在 内存中 是MySQL 调优的一个重要方面。

Buffer Pool LRU Algorithm
buffer pool 使用 LRU的变体 来管理。当需要空间 来增加新的page 到 buffer pool 时， 最近最少使用的 page 被逐出，并将新page添加到 list的 middle。 midpoint insertion strategy(中点插入策略) 将 list 视为 2个 sublist：

1. 头部，是 最近访问的 新(年轻)页面 的sublist
2. 尾部，是 最近较少访问的 旧page 的 sublist



这个算法 将经常使用的 page 保留在 新sublist中。 旧sublist 包含访问较少的page，这些page是 逐出的 候选者。

默认下，算法运算如下：

1. buffer pool 的 3/8 用于 旧sublist
2. list的 midpoint 是 新sublist 和 就sublist 相交的地方。
3. 当 InnoDB 将 page 插入到 buffer pool，最初会插入到 midpoint(旧sublist的 head)。一个page可以被读取，因为 它是用户启动的操作 所 需要的 或者 作为InnoDB自动执行的预读取操作的 一部分。
4. 访问 旧sublist中的 page，会使 page 变 年轻，移动到 新sublist的 头部。如果page 是因为 用户的操作 而需要被读取的，则 first access会立即发生，并且 page 变得 年轻。 如果 由于 预读取操作 而读取了 page，则 first access 不是立刻发生，并且 可能在 页面 被逐出之前 根本不会发生。
5. 随着数据库运行，buffer pool 中的 没有被 访问过的 page 会 逐渐 向 列表尾部 移动 而 老化。新旧子列表中的 page 都会随着 其他page 的更新 而老化。旧sublist的 page 也会 随着 midpoint 的插入 而 老化。最终 未使用的 page 到达 旧sublist 的尾部 并被 驱逐。

默认下，query读取的 page 会立即移动到 新sublist中，意味着 它们在 buffer pool 中 停留

的时间 更长。例如，执行 mysqldump 操作 或 不带where的select 的表扫描 可以将 大量数据 带入 buffer pool 并 驱逐 等量的 旧数据，即使这些 新数据 不会被 再次使用。 类似地，由预读后台线程 加载 且 仅访问一次的 页面被 移动到 新sublist 的 头部。 这些情况，经常将 那些经常使用的page 被推到 旧sublist中，甚至它们会被驱逐。

InnoDB Standard Monitor output 包含 BUFFER POOL AND MEMORY 章节 中有 关于 buffer pool LRU 算法的 几个字段。

Buffer Pool Configuration
你可以配置buffer pool 的各个方面 来提高性能

1. 理想情况下，你将 buffer pool 的大小设置得 尽可能大，但要为 服务器上 其他进程 留出 足够的 内存 来运行， 而不会出现 过多的 OS的 paging。 buffer pool 越大，InnoDB 就越像 内存数据库， 从磁盘读取一次数据，然后在后续读取期间 从 内存中 访问数据。
2. 在具有 足够内存的 64位OS上，你可以将 buffer pool 拆分为 多个部分，以尽量减少 并发操作 之间的 内存 结构 contention (争吵，争论，竞争)
3. 你可以将经常访问的 数据保留在 内存中，而不管 突然的高峰 会将 不经常访问的 数据导入 buffer pool。
4. 你可以控制 执行预读 请求的 方式 和时间， 以异步将 页面 预取 到 buffer pool 中，以应对 即将到来的请求。
5. 你可以控制 什么时候进行后台flushing ，是否根据 工作负载 动态 调整 刷新速率。
6. 你可以配置 InnoDB， 如何保存 当前 buffer pool 状态 以 避免 server 重启后 的 长时间 预热。

Monitoring the Buffer Pool Using the InnoDB Standard Monitor
InnoDB Standard Monitor output，能通过 show engine innodb status 来看到。 buffer pool 的指标 在 BUFFER POOL AND MEMORY 部分中。

```
----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 2198863872
Dictionary memory allocated 776332
Buffer pool size   131072
Free buffers       124908
Database pages     5720
Old database pages 2071
Modified db pages  910
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 4, not young 0
0.10 youngs/s, 0.00 non-youngs/s
Pages read 197, created 5523, written 5060
0.00 reads/s, 190.89 creates/s, 244.94 writes/s
Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not
0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read
ahead 0.00/s
LRU len: 5720, unzip_LRU len: 0
```

I/O sum[0]:cur[0], unzip sum[0]:cur[0]

下标描述了 InnoDB standard monitor 的 buffer pool 的 指标。

每秒平均值 是指 上次 打印 到 现在 这段时间内的 每秒平均值。

InnoDB Buffer Pool Metrics

Name	Description
Total memory allocated	The total memory allocated for the buffer pool in bytes.
Dictionary memory allocated	The total memory allocated for the InnoDB data dictionary in bytes.
Buffer pool size	The total size in pages allocated to the buffer pool.
Free buffers	The total size in pages of the buffer pool free list.
Database pages	The total size in pages of the buffer pool LRU list.
Old database pages	The total size in pages of the buffer pool old LRU sublist.
Modified db pages	The current number of pages modified in the buffer pool.
Pending reads	The number of buffer pool pages waiting to be read into the buffer pool.
Pending writes LRU	The number of old dirty pages within the buffer pool to be written from the bottom of the LRU list.
Pending writes flush list	The number of buffer pool pages to be flushed during checkpointing.
Pending writes single page	The number of pending independent page writes within the buffer pool.
Pages made young	The total number of pages made young in the buffer pool LRU list (moved to the head of sublist of “new” pages).
Pages made not young	The total number of pages not made young in the buffer pool LRU list (pages that have remained in the “old” sublist without being made young).
youngs/s	The per second average of accesses to old pages in the buffer pool LRU list that have resulted in making pages young. See the notes that follow this table for more information.
non-youngs/s	The per second average of accesses to old pages in the buffer pool LRU list that have resulted in not making pages young. See the notes that follow this table for more information.
Pages read	The total number of pages read from the buffer pool.
Pages created	The total number of pages created within the buffer pool.
Pages written	The total number of pages written from the buffer pool.
reads/s	The per second average number of buffer pool page reads per second.

creates/s	The average number of buffer pool pages created per second.
writes/s	The average number of buffer pool page writes per second.
Buffer pool hit rate	The buffer pool page hit rate for pages read from the buffer pool vs from disk storage.
young-making rate	The average hit rate at which page accesses have resulted in making pages young. See the notes that follow this table for more information.
not (young-making rate)	The average hit rate at which page accesses have not resulted in making pages young. See the notes that follow this table for more information.
Pages read ahead	The per second average of read ahead operations.
Pages evicted without access	The per second average of the pages evicted without being accessed from the buffer pool.
Random read ahead	The per second average of random read ahead operations.
LRU len	The total size in pages of the buffer pool LRU list.
unzip_LRU len	The length (in pages) of the buffer pool unzip_LRU list.
I/O sum	The total number of buffer pool LRU list pages accessed.
I/O cur	The total number of buffer pool LRU list pages accessed in the current interval.
I/O unzip sum	The total number of buffer pool unzip_LRU list pages decompressed.
I/O unzip cur	The total number of buffer pool unzip_LRU list pages decompressed in the current interval.

youngs/s 指标 仅适合于 old page。它基于 page access 次数，如果一个page 有多次 access，都被计算在内。 如果在没有发生 大的scan时 看到 非常低的 youngs/s ，考虑 减少 延迟时间 或 增加 用于 旧sublist 的 buffer pool 的百分比。 增加百分比 会使 旧sublist 变大， 从而 使 该 sublist 的 page 移动到 尾部 所需的 时间更长， 这增加了 再次访问这些页面 并使其变 年轻的 可能性。

non-youngs/s 仅适用于 old page。 基于页面访问次数，多次访问 都计算。如果在 执行 large table scan 时，没有看到 更高 的 non-youngs/s 值 (以及更高的 youngs/s 值)， 增加delay value.

young-making rate 适用于 所有的buffer pool page access ，不仅仅是 old sublist 的 page。 young-making 率 和 not 率 通常不加总到 buffer pool hit rate。old sublist 的 page hit 会导致 page 移动到 new sublist， 但 new sublist 中的 page hit 会导致 page 移动到 list的 头部，前提是 它们和 头部 有一定距离。

not(young-making rate) 是 由于 未满足 innodb_old_blocks_time 定义的延迟，或 由于 在 newlist 中page hit 但是没有触发移动page 到 头部。 这个 rate 是所有 buffer pool page access，不仅仅是 old sublist 的 page access。

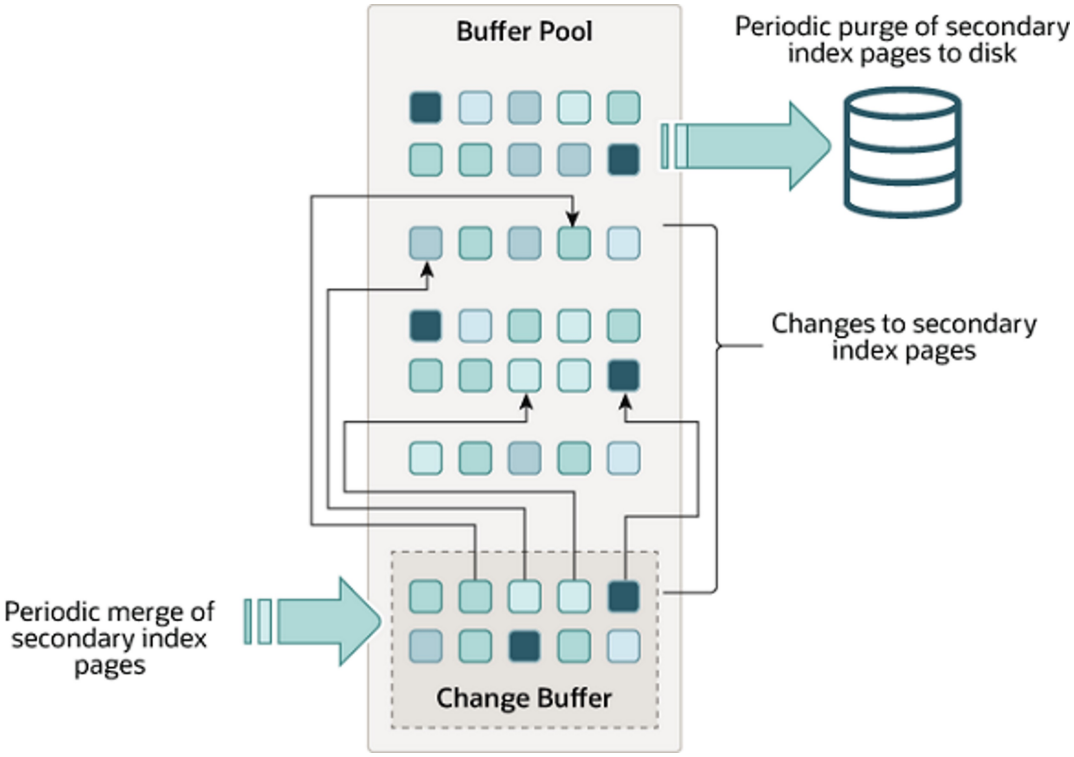
buffer pool server status variables 和 innodb_buffer_pool_stats 表 提供了 许多 和 InnoDB 标准监视器 输出 中 相同 的缓冲池 指标。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-change-buffer.html

15.5.2 Change Buffer

change buffer 是一个 特殊的 数据结构， 用于 cache 二级索引page 的 change，当 这些 page 不在 buffer pool 中时。
被buffer的change（它们可能来自 insert，update，delete操作）稍后 会被merge，当 page 被其他 read 操作 读取到 buffer pool 时。



与聚集index不同， 二级index通常是 非唯一的， 并且 insert 二级index 的顺序 相对 随机。 类似地，delete 和 update 可能影响 index tree中 不相邻的 二级index page。稍后合并 cached change，当受到影响的 页面 被其他 操作 读入 buffer pool时 ； ； 避免了 将二级index page 从 磁盘读入 buffer pool 所需的 大量 随机 IO

当系统空闲 或 在 slow shutdown 期间 发生的 purge 操作，会将 updated index page 写入磁盘。 与 将每个值立即写入 磁盘相比， purge操作 可以 更有效地 为一系列 index值 写入 disk block。

change buffer merging 可能花费 几个小时，如果有许多 受影响的 row 和 二级索引需要更新

时。在此期间， 磁盘 IO 会增加，这可能导致 磁盘上的 查询 显著变慢。 在 事务commit 后，甚至在 服务器 关闭 和重启后， change buffer merging 也可能 继续 发生。

内存中，change buffer 占据了 buffer pool 的一部分。在磁盘上，change buffer 是 system tablespace 的一部分，index change 被buffer 当 数据库服务器关闭时。

change buffer 中 cache 的 数据的类型 由 innodb_change_buffering 变量 控制。 你还可以配置 最大change buffer size。

如果index 包括 降序index column 或 主键包含 降序index column， 而二级index 不支持 change buffering

For answers to frequently asked questions about the change buffer, see Section A.16, “MySQL 8.0 FAQ: InnoDB Change Buffer”.

Configuring Change Buffering
当对表 执行 insert,update,delete操作时， index column 的值 (尤其是 secondary key 的值) 通常处于 未排序的 顺序， 需要 大量 IO 才能 使得 二级index 保持最新。 当相关page 不在 buffer pool中时，change buffer cache 对二级index 的 change，而不是 立即从磁盘读取page ，这样可以避免 昂贵的 IO操作。 当page 被加载到 buffer poll中， buffered change 被merge， 更新后的page 稍后 flush 回 disk。 InnoDB 主线程 merge buffered change 当服务器空闲时，以及在 slow shutdown 期间。

因为它可以减少 磁盘 读取和写入， 所以 change buffering 对 IO密集型的 工作负载 最有价值。例如，具有大量 DML操作 (如 批量insert) 的 应用程序 受益于 change buffering。

当时，change buffer 占用了 buffer pool 的一部分， 从而减少了 可用于 cache page 的 内存。如果工作集 几乎适合 buffer pool (。。应该是指数据能全部加载到内存的buffer pool 中)，或 你的表 具有 相对较少的 二级index，则 禁用 change buffer 可能很有用。

innodb_change_buffering 变量 控制 innodb 执行 change buffering 的程度。 你可以 启用 或 禁用 buffering for insert,delete操作(index 被标记为 删除)， purge操作(index 被物理删除)。 innodb_change_buffering 默认值是 all。

innodb_change_buffering 可能的值：

- all， 默认值，包括 buffer insert,delete-marking，pruge
- none，不 buffer 任何 操作
- inserts， buffer insert 操作
- deletes，buffer delete-marking 操作
- changes， buffer insert 和 delete-marking
- purges, buffer 后台发生的 物理删除操作。

你可以在 MySQL option file （my.conf, or my.ini） 中 设置 innodb_change_buffering变量， 或者 仅用 set global 语句 来 动态更改它， 这需要 足够的权限 来设置 全局系统变量。 更改配置 会影响 新操作的 buffering， 已buffer 的 条目的 merge 不受影响。

Configuring the Change Buffer Maximum Size

innodb_change_buffer_max_size 变量 允许 将 change buffer 的 最大大小 设置为 buffer pool 大小的 百分比。 默认下，innodb_change_buffer_max_size 是25. 最大值是 50。

考虑在具有大量insert，update，delete，导致 change buffer merging 的速度跟不上 new change buffer，导致 change buffer 达到 最大大小限制 的 mysql上 增加 innodb_change_buffer_max_size，

考虑在 用于报告 的 静态数据的 MySQL 服务器上 减小 innodb_change_buffer_max_size，或者 如果 change buffer 消耗了过多的 内存(这些内存 与 buffer pool 共享)，导致 page 比 预期 更早地 从 buffer pool 中 老化。

使用 具有代表的 工作负载 测试 不同的 配置 以确定 最佳配置，
innodb_change_buffer_max_size 是动态的，可以在 允许时 修改。

Monitoring the Change Buffer

下面的选项可以用于 change buffer monitoring：

1. innodb standard monitor output 包含 change buffer status信息。要查看 监控数据，使用 show engine innodb status 语句

   ```
   mysql> SHOW ENGINE INNODB STATUS\G
   ```

   change buffer status 信息 位于 INSERT BUFFER AND ADAPTIVE HASH INDEX 标题下，类似下面的内容：

   ```
   -------------------------------------
   INSERT BUFFER AND ADAPTIVE HASH INDEX
   -------------------------------------
   Ibuf: size 1, free list len 0, seg size 2, 0 merges
   merged operations:
    insert 0, delete mark 0, delete 0
   discarded operations:
    insert 0, delete mark 0, delete 0
   Hash table size 4425293, used cells 32, node heap has 1 buffer(s)
   13577.57 hash searches/s, 202.47 non-hash searches/s
   ```

2. information_schema.innodb_metrics 表 提供了 在 InnoDB 标准监控输出 中 找到的 大多数 数据点 及其他数据。 要查看 change buffer指标 和 每个指标的 描述，请使用：

   ```
   mysql> SELECT NAME, COMMENT FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE
   NAME LIKE '%ibuf%'\G
   ```

3. infomation_schema.innodb_buffer_page 表 提供了 有关 buffer pool 中 每个 page 的 metadata，包括 change buffer index 和 change buffer bitmap pages。 change buffer page 通过 PAGE_TYPE标识。 IBUF_INDEX 是 page tpye，标识 change buffer index pages, IBUF_BITMAP 是page type，表示 change buffer bitmap pages。

   查询 innodb_buffer_page 会 占用 显著的 性能开销。 为了避免影响性能，请在 测试环境重现问题，然后再 测试环境 进行查询。

   例如，你可以查询 innodb_buffer_page 表 来确定 ibuf_index 和 ibuf_bitmap page 占 buffer pool 的总页数 的 百分比

   ```
   mysql> SELECT (SELECT COUNT(*) FROM INFORMATION_SCHEMA.INNODB_BUFFER_PAGE
          WHERE PAGE_TYPE LIKE 'IBUF%') AS change_buffer_pages,
   ```

```
        (SELECT COUNT(*) FROM INFORMATION_SCHEMA.INNODB_BUFFER_PAGE) AS
total_pages,
        (SELECT ((change_buffer_pages/total_pages)*100))
        AS change_buffer_page_percentage;
+--------------------+-------------+-------------------------------+
| change_buffer_pages | total_pages | change_buffer_page_percentage |
+--------------------+-------------+-------------------------------+
|                  25 |        8192 |                        0.3052 |
+--------------------+-------------+-------------------------------+
```

4. performance schema 为 高级 性能监控 提供了 change buffer mutex wait instrumentation。要查看 change buffer instrumentation ，使用下面的query：

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%wait/synch/mutex/innodb/ibuf%';
+-----------------------------------------------------+---------+-------+
| NAME                                                | ENABLED | TIMED |
+-----------------------------------------------------+---------+-------+
| wait/synch/mutex/innodb/ibuf_bitmap_mutex           | YES     | YES   |
| wait/synch/mutex/innodb/ibuf_mutex                  | YES     | YES   |
| wait/synch/mutex/innodb/ibuf_pessimistic_insert_mutex | YES   | YES   |
+-----------------------------------------------------+---------+-------+
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html

15.5.3 Adaptive Hash Index

自适应hash index 使 innodb 能够在（适当的工作负载，足够内存的）系统上 执行得更像 内存数据库，而不会牺牲 事务功能 和 可靠性。
adaptive hash index 由 innodb_adaptive_hash_index 启用，或在 服务器启动时 通过 --skip-innodb-adaptive-hash-index 来关闭。

根据观察到的 搜索模式， 使用index 的前缀构建 hash index。前缀可以是 任意长度，也可能只有 B tree中的 一些值 出现在 hash index中。hash index 是 针对 经常访问的 index page 的需求 而构建的。

如果一个表 几乎完全适合内存，hash index 通过 启用 任何元素的 直接lookup 来加速查询，将index值 转换为 一种指针。 InnoDB 有一个 监控index search 的机制，如果 InnoDB 注意到 查询可以从 构建hash index 中受益， 它会自动这么做。

对于某些工作负载，hash lookup 的 加速 大大超过了 监控 index search 和 维护 hash index 结构的 额外工作。 在繁重的工作负载下，对 自适应hash index 的访问 有时成为 竞争的来源。 使用 like 和 % 通配符 的查询 也往往 不会收益。 对于不能从 自适应hash index 中受益的 工作负载，关闭它 可以减少不必要的 性能开销。 因为很难预计 自适应hash index 是否适应 特定系统和工作负载， 请考虑 在启用 和 禁用 它的情况下 运行 基准测试。

自适应hash index 是 分区的。 每个index 都被绑定到 一个特定的 分区，每个分区都有一个单独的 latch(锁存器，门闩) 保护。 分区由 innodb_adaptive_hash_index_parts 控制，默认8， 最大可以设置为 512。

你可以在 show engine innodb status 输出的 semaphores 部分 监视 自适应hash index 的使用 和 竞争。 如果有 许多线程 在 btr0sea.c 中创建的 rw-latch 上等待， 请考虑增加自适应 hash index 分区的 数量 或 禁用 自适应hash index。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-redo-log-buffer.html

15.5.4 Log Buffer

log buffer 是 保存 要写入 磁盘 的 日志文件的 内存区域。 log buffer 大小 由 innodb_log_buffer_size 变量 定义。 默认大小 16mb。
log buffer 的内容 会 定期刷新到 磁盘。 大型log buffer 是的 大型事务能够运行，而无需在事务 提交之前 将 undo log 写入磁盘。 因此， 如果你有 update，insert，delete 许多行的 操作 的事务，增加 log buffer 来节约 磁盘IO

innodb_flush_log_at_trx_commit 变量 控制 log buffer 的内容 如何写入 和 刷新到 磁盘。
innodb_flush_log_at_timeout 控制 log flush 频率

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-on-disk-structures.html
15.6 InnoDB On-Disk Structures

15.6.1 Tables
15.6.2 Indexes
15.6.3 Tablespaces
15.6.4 Doublewrite Buffer
15.6.5 Redo Log
15.6.6 Undo Logs

15.6.1 Tables

15.6.1.1 Creating InnoDB Tables
15.6.1.2 Creating Tables Externally
15.6.1.3 Importing InnoDB Tables
15.6.1.4 Moving or Copying InnoDB Tables
15.6.1.5 Converting Tables from MyISAM to InnoDB
15.6.1.6 AUTO_INCREMENT Handling in InnoDB

---
https://dev.mysql.com/doc/refman/8.0/en/using-innodb-tables.html

15.6.1.1 Creating InnoDB Tables

InnoDB表 使用 create table 语句来创建：

```
CREATE TABLE t1 (a INT, b CHAR (20), PRIMARY KEY (a)) ENGINE=InnoDB;
```

如果InnoDB被定义为 默认存储引擎，则不需要 engine=innodb 子句。
用下面的语句 来查看 MySQL Server 实例的 默认 存储引擎：

```
mysql> SELECT @@default_storage_engine;
+--------------------------+
| @@default_storage_engine |
+--------------------------+
| InnoDB                   |
+--------------------------+
```

默认情况下，InnoDB表 是在 file-per-table 表空间中创建的。 要在 InnoDB system 表空间中创建 表， 请在 创建表 之前 禁用 innodb_file_per_table 变量。 要在 通用 表空间中创建 InnoDB 表，请使用 create table .. tablespace 语法。

Row Formats
InnoDB表的 row format 决定了 它的row 在 磁盘上的 物理存储方式。
InnoDB支持 4种 row format：REDUNDANT,COMPACT,DYNAMIC,COMPRESSED。 默认 dynamic。
innodb_default_row_format 定义了 默认的 row format， 还可以在 create table 或 alter table 语句的 row_format 选项中显示定义 row format。

Primary Keys
建议你 在创建每个表时 定义一个主键。选择主键列时，选择具有以下特征的列：

1. 会被 最重要的query 用到的 column
2. 永远不会 空的 列
3. 永远不会有 重复值的 列
4. 插入后 很少更改的 列。

例如，在包含人员信息的表中，你不会在 (firstname,lstname) 上创建 主键，因为可以有 多个人 具有 相同的姓名，姓名列可能留空，有时人们会改名字。 由于有如此多个 约束，通常没有一组 明显的 列可以用作 主键， 因此你创建 一个具有 数字 id 的新列 作为 主键的 全部 或部分。 你可以 声明一个 自动增量列，以便在插入row时 自动填充 升序值：

```
# The value of ID can act like a pointer between related items in different
tables.
CREATE TABLE t5 (id INT AUTO_INCREMENT, b CHAR (20), PRIMARY KEY (id));

# The primary key can consist of more than one column. Any autoinc column must
come first.
```

```
CREATE TABLE t6 (id INT AUTO_INCREMENT, a INT, b CHAR (20), PRIMARY KEY
(id,a));
```

尽管表可以在 不定义主键的 情况下 正常工作， 但 主键涉及性能的 许多方面，并且 对于 任何 大型或 经常使用的 表 来说 都是 至关重要的 设计方面。 建议你 始终 在 create table 中 指定 主键。 如果 创建表，加载数据，然后运行alter table 添加主键，则 该操作 比 创建表 是 定义 主键 要慢很多。

Viewing InnoDB Table Properties
要查看 innodb 表的属性， 请使用 show table status

```
mysql> SHOW TABLE STATUS FROM test LIKE 't%' \G;
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Dynamic
           Rows: 0
 Avg_row_length: 0
    Data_length: 16384
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2021-02-18 12:18:28
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_0900_ai_ci
       Checksum: NULL
 Create_options:
        Comment:
```

你还可以通过 查询 InnoDB 的 information schema 系统表 来访问 InnoDB 表属性：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME='test/t1' \G
*************************** 1. row ***************************
     TABLE_ID: 1144
         NAME: test/t1
         FLAG: 33
       N_COLS: 5
        SPACE: 30
   ROW_FORMAT: Dynamic
ZIP_PAGE_SIZE: 0
   SPACE_TYPE: Single
 INSTANT_COLS: 0
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-create-table-external.html

15.6.1.2 Creating Tables Externally

在 外部创建 Innodb 表（也就是说，在 data directory 之外创建表。） 有不同的原因。 可能包括 空间管理，IO优化，或将表放置在 具有特定性能 或容量特征的 存储设备上。
。。SSD。。

支持下面的方式 来创建 外部表
    Using the DATA DIRECTORY Clause
    Using CREATE TABLE ... TABLESPACE Syntax
    Creating a Table in an External General Tablespace

1.
```
CREATE TABLE t1 (c1 INT PRIMARY KEY) DATA DIRECTORY = '/external/directory';
    mysql> SELECT @@datadir,@@innodb_data_home_dir,@@innodb_directories;
```

2.
```
mysql> CREATE TABLE t2 (c1 INT PRIMARY KEY) TABLESPACE = innodb_file_per_table
       DATA DIRECTORY = '/external/directory';
```

3.
You can create a table in a general tablespace that resides in an external directory.

。。基本都跳了。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-table-import.html

15.6.1.3 Importing InnoDB Tables

本节介绍 如何使用 transportable tablespaces 功能 导入表。该功能允许 导入 位于 file-per-table 表空间的 表， 分区表 或 单个表分区。
导入表 有许多理由：

1. 在非生产的 MySQL服务器上 允许 report，以避免 生产服务器 上额外的负担
2. 将数据 复制到新的 副本服务器
3. 从备份 的表空间文件 中 恢复表
4. 比 导入dump文件更快 (导入dump文件需要 重新插入数据 和 重建idnex)
5. 将数据 移动到 更适合 你的存储要求的存储介质 的服务器上， 例如你可以将繁忙的表移动到SSD，将大型表移动到 大容量的 HDD。

The Transportable Tablespaces feature is described under the following topics in this section:
    Prerequisites
    Importing Tables

Importing Partitioned Tables
Importing Table Partitions
Limitations
Usage Notes
Internals

。。跳过

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-migration.html

15.6.1.4 Moving or Copying InnoDB Tables
本节介绍 将 部分或全部 InnoDB 表 移动或复制 到不同的服务器 或instance 的技术。例如，你可以将 整个MySQl实例 移动到 更大，更快的服务器； 你可以将整个MySQL实例 clone 到 新的副本服务器； 你可以将 单个表 复制到 另一个实例 以开发 和 测试 应用程序，或复制到 数据仓库服务器 以生产报告。

在window上， InnoDB 始终在内部 以小写形式存储 数据库和 表名。要将二进制格式的 数据库从Unix 移动到 window 或从 window移动到 unix，请使用 小写名称 创建所有的数据库和表。一个方便的方法是创建 任何数据库 或表之前 将以下行 添加到 my.cnf 或 my.ini 的 [mysqld] 部分：

```
[mysqld]
lower_case_table_names=1
```

注意：禁止使用 与 初始化服务器时 使用的 设置不同的 lower_case_table_names 来启动服务器。

移动或复制 InnoDB 表的 技术包括：
Importing Tables
MySQL Enterprise Backup
Copying Data Files (Cold Backup Method)
Restoring from a Logical Backup

。。跳

---

https://dev.mysql.com/doc/refman/8.0/en/converting-tables-to-innodb.html

15.6.1.5 Converting Tables from MyISAM to InnoDB

如果你想将 MyISAM 表 转为InnoDB 以获得 更好的 可靠性 和 可扩展性，请在转换前 查看以下指南 和 提示。
Adjusting Memory Usage for MyISAM and InnoDB
Handling Too-Long Or Too-Short Transactions

Handling Deadlocks
Storage Layout
Converting an Existing Table
Cloning the Structure of a Table
Transferring Data
Storage Requirements
Defining Primary Keys
Application Performance Considerations
Understanding Files Associated with InnoDB Tables

。。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-auto-increment-handling.html

15.6.1.6 AUTO_INCREMENT Handling in InnoDB

InnoDB 提供了 一种 可配置 的锁定机制， 可以显著提高 向 具有 auto_increment 列的 表添加行的 sql 语句的 可伸缩性 和 性能。要将 auto_increment 机制 和 InnoDB表一起使用， 必须将 auto_increment 列 定义为 某个 索引的 第一列 或 唯一列， 以便可以 对 表执行 等效的 索引 select max(ai_col) 查找 以 获得 最大值列值。index 不需要是 primary key 或 unique， 但为了 避免 auto_increment 列中的重复值，建议使用 这些 index类型。

本节描述 auto_increment 的 lock mode， 不同 lock mode 的 使用含义，InnoDB 如何初始化 auto_increment 计数器。

InnoDB AUTO_INCREMENT Lock Modes

本节介绍 生成 auto-increment value 的 lock mode，以及每种 lock mode 如何影响复制。auto-increment lock mode 在 启动时 使用 innodb_autoinc_lock_mode 变量。

下面是 用于描述 innodb_autoinc_lock_mode 设置：

1. "INSERT-like" 语句
   所有 生成new row 的语句，包括 insert,insert..select,replace,replace..select,load data。 包含了 simple-inserts, bulk-inserts, mixed-mode 插入

2. "Simple inserts"
   这些语句 可以 预先确定要插入的row 的数量 (在最初处理语句时)。这包括 没有嵌套subquery 的 insert 和 replace， 但没有 insert .. on duplicate key update

3. "Bulk inserts"
   这些语句 预先不知道 要插入的 row数量 (以及所需的 auto-increment value)。包括 insert..select, replace..select, load data， 但不包括 纯 insert。InnoDB 在处理每行时，为 auto_increment 列分配一个 新值。

4. "Mixed-mode inserts"
   这些是 simple insert 语句 且 指定了 一些(不是all) 新row 的 自动增量值。下

面的例子，c1是 auto_increment 列：

```
INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'),
(NULL,'d');
```

另一种类型的 mixed-mode insert 是 insert .. on duplicate key update，最坏情况下，它实际上是一个 insert 后跟 一个update， 其中为 auto_increment 列的分配的值 可能会被使用 也可能不会被使用， 在 update 阶段。

innodb_autoinc_lock_mode 有3个可能的配置。 分别是 0,1,2，对应了traditional,consecutive,interleaved (传统，连续，交错) 的 lock mode。
从MySQL 8.0开始， interleaved lock mode (innodb_autoinc_lock_mode=2) 是 默认的， 在8.0之前，consecutive lock mode (xxxx=1) 是默认的。

MySQL 8.0中， interleaved lock mode 的默认设置 反应了 默认复制类型的 变化 (从 基于语句的复制 到 基于行的复制)。 基于语句的复制 需要 consecutive auto-inc lock mode 来确保 给定的sql语句序列 以 可预测 和 可重复的 顺序 分配 自增值。而 基于行的复制 对sql语句的 执行顺序 不敏感。

1. innodb_autoinc_lock_mode = 0 (traditional lock mode)
   传统锁模式 提供了 与引入 innodb_autoinc_lock_mode 变量之前相同的行为。由于语义上可能存在差异，提供 0模式 是为了 向后兼容，性能测试，和 解决 "mixed-mode inserts" 的问题。
   这种模式下，所有的 "insert-like"语句都会 获得一个 特殊的 table-level 的 auto-inc lock 用于插入 到具有 auto-increment 列的表中。 此锁 通常保持到 语句的末尾 (而不是 事务的末尾)， 以确保 为给定的 insert 语句序列以 可预测和 可重复的 顺序分配 自动递增值，并确保 分配给任何语句的 自增值都是连续的。
   在 statement-based replication 的情况下，这意味着 当 在 副本服务器上 复制sql 语句时，自动增量列 使用 与 源服务器 相同的值。 多个insert语句的 执行 结果 是确定性的， 副本复制了 与源上相同的数据。 如果 多个insert语句生成的 auto-inc 的值是 interleaved，那么 2个并发的 insert 语句 的结果 是不确定的，并且 无法使用 基于语句的复制 可靠地 传播到 副本服务器。

   为了更清晰理解，考虑使用了下面表的 例子

   ```
   CREATE TABLE t1 (
     c1 INT(11) NOT NULL AUTO_INCREMENT,
     c2 VARCHAR(10) DEFAULT NULL,
     PRIMARY KEY (c1)
   ) ENGINE=InnoDB;
   ```

   假设有2个事务在运行，每个事务都将 行插入到 具有 auto_increment 列的表中。 一个事务 使用 插入1000行的 insert .. select 语句， 另一个 事务 使用了 插入一行 的简单 insert 语句：

   ```
   Tx1: INSERT INTO t1 (c2) SELECT 1000 rows from another table ...
   Tx2: INSERT INTO t1 (c2) VALUES ('xxx');
   ```

   InnoDB 无法提前知道 从tx1 中的 insert 语句中的 select 检索到 多少行，并且 随着语句的执行，它一次分配一个 自增量。使用 table-level lock,一直到 语句结束，一次只能执行一次t1表的insert语句，不同的语句 生成的 自增值 不会 交错。 tx1 的语句生成的 自增值 是 连续的， 并且 tx2中的insert 使用的 自增值 小于 或 大于 tx1的所

有值，取决于哪个事务先执行。

只要sql语句 从 binary log 中重放时 (当使用 基于语句的复制时，或在 恢复场景中) 以相同的顺序执行，结果与 tx1 和 tx2 首次运行时的 结果相同。因此，表级锁 一直 保持到 语句结果，使得 自增值的 insert语句 可以安全地 用 基于 语句的复制。 但是，当多个事务 同时执行插入语句时， 这些 表级锁 会 限制 并发性和 可伸缩性。

。。binary log 中 也保持了 事务的 先后？ 上(上)段的最后一句。 保持了事务的先后的话， 多事务插入 还是得排队啊， 和 表级锁 有什么关系。

。。不，就是 自增列的 值 是 实时的， 就是 自增列的值 可以在 一开始就设置，然后、、、反正，事务和这里无关， 事务是ACID， 这里讨论的是 自增值的 赋值。 这里通过表级auto-inc 锁 来保证 自增值的 赋值 是 相同的。。 但是 我还是不清楚 这个 binary log 保存了什么东西， 至少得保存 谁先获得 表级锁吧。 不然 自增值 是不能复现的。

在前面的例子中，如果没有表级锁，则用于 tx2 的 insert 的自增列的值 取决于 语句执行的时间。 如果 tx2 的insert 在 tx的insert 执行时 执行 (而不是在tx1 之前或 之后)， 则 2个 insert 语句 分配的 自增量 是 不确定的，并且可能 因运行而异。

在 consecutive(连续) lock mode 下，InnoDB 可以避免 对 预先知道行数的 "simple-insert" 语句 使用 表级 auto-inc 锁，并且 仍然为 基于语句的 复制 保留 确定性执行和 安全性。

如果你不使用 binary log 来 replay sql语句 来作为 revocery 或 replication 的一部分，那么可以使用 interleaved lock mode 来 消除 所有 表级 auto-inc 锁的使用， 以获得 更高的 并发性 和 性能， 但是 代价是 允许 分配给一个语句的 自增量 编号 中间出现 间隙， 并且 由于并发执行的语句，所以有 交错分配的 自增量。

2. innodb_autoinc_lock_mode = 1 (“consecutive” lock mode)
这种模式下，"bulk insert" 使用特殊的 auto-inc 表级锁 并持有它直到 语句结束。这适用于 所有的 insert ..select, replace .. select , load data 。一次只能执行一个 持有 auto-inc 锁的 语句。如果 批量插入 的 源表和目标表 不同，则在对 从源表中 选择的第一行 进行 共享锁定后， 对 目标表进行 auto-inc 锁定。如果批量插入的 源表和目标表 是同一个表，则在 对所有选定行 进行 共享锁后， 再使用 auto-inc 锁。

"simple insert" (预先知道要插入的row 数量) 通过在 mutex(轻量级锁) 的控制下 获得 所需数量的 自增值 来避免 表级 auto-inc 锁， 这个mutex 只在 allocate 过程中 被占用，而不是 直到语句完成。 不使用表级auto-inc锁， 除非另一个事务 持有了 auto-inc 锁。 如果 另一个事务 持有 auto-inc锁，则 simple insert 会等待 auto-inc 锁，就好像它是 "bulk insert" 一样。

这种 lock mode 可以保证， 在 预先不知道 行数 的insert 语句(以及 在语句处理时 分配 自增量) 的情况下，由 任何 "insert-like" 分配的 所有 自增量 都是连续的， 并且操作 对于 基于语句的 复制 是安全的。

简而言之，这种lock mode 显著提高了 可伸缩性，同时 可以安全地 用于 基于语句的 复制。 此外，与"传统" 锁定模式 一样，任何 给定语句 分配的 自增量 是 连续的。 对于任何使用 自增量的语句，和 传统模式相比， 语义没有变化，但有一个重要例外。

例外是"mixed-mode inserts"， 其中用户 为 多行 "simple insert" 中的 某些 (但不是全部) 行 提供 auto_increment 列的 显示值。 对于这种insert， innodb 分配的 自增

量 的个数 多于 要插入的 行数。 但是，所有自动分配的 值 都是 连续生成的 (因此高于) 最近执行的 前一条语句 生成的自动增量值。。多余的自增量被丢弃。
。。就是分配了 100个， 但是可能只用了 80个，并且用的 是 1-80的。 81-100的 被丢弃了。

3. innodb_autoinc_lock_mode = 2 (“interleaved” lock mode)
   在这个lock mode，没有 "insert-like" 语句使用 表级 auto-inc 锁，多个语句可以同时执行。这是最快的 且 最具可伸缩性的 lock mode，但是在 从 binary log replay sql语句时，使用基于 语句的 复制 或 恢复场景时，它并不安全。
   这个lock mode下， 自增量 保证 在所有 并发执行的 "insert-like" 的语句中 是唯一的且 单调递增的。但是，由于多个语句可以同时 生成自增量 (即，数字的分配在语句之间交错)，为任何给定语句 插入的 row 生成的 值 可能不会 连续的。
   如果执行的 唯一语句是 "simple insert"，其中要插入的 行数 是提前知道的， 那么除了"mixed-mode inserts" 之外， 为单个语句生成的 自增量 是没有间隙的。 但是， 当执行 "bulk insert" 时， 任何 给定语句分配的 自增量 可能存在间隙。

InnoDB AUTO_INCREMENT Lock Mode Usage Implications
Using auto-increment with replication
   如果你使用基于 语句的 复制， 请将 innodb_autoinc_lock_mode 设置为0 或1，并在 源和 它副本 上 使用 相同的值。 如果使用 2模式， 或 源和副本 不使用 相同的 lock mode， 则不能确保 副本的 自增量 和 源上的值 相同。
   如果你使用的是 基于 row 或 mixed-format 的复制，所有的 auto-increment lock mode 是安全的， 因为 基于row 的复制 对 sql语句执行顺序 不敏感 (并且 混合格式 使用 基于row 的复制 对那些 基于语句的复制是不安全的 复制)

“Lost” auto-increment values and sequence gaps
   所有lock mode (0,1,2) 中，如果 生成 自增量的 事务 回滚，则 这些 自增量 会 丢失。一旦 为 自增值 生成了值， 无论"insert-like" 语句是否 完成， 以及 包含的 事务是否 回滚， 都无法回滚。这种 丢失的 值 不会被重用。 因此，存储在 表的 auto_increment 列 中的 值 可能存在 间隙。

Specifying NULL or 0 for the AUTO_INCREMENT column
   所有lock mode, 如果用户 在 insert 时 为 auto_increment 的列 指定 null 或 0，innoDB会将该row 视为 未指定该值 并为其生成 一个 新值。

Assigning a negative value to the AUTO_INCREMENT column
   所有lock mode, 如果 将 负数 分配给 auto_increment 列， 则 自增量机制 的行为 是未定义的。。。。

If the AUTO_INCREMENT value becomes larger than the maximum integer for the specified integer type
   所有lock mode, 如果 自增量 上溢， 行为未知

Gaps in auto-increment values for “bulk inserts”
   将 innodb_autoinc_lock_mode 设置为 0 或 1 (traditional or consecutive)， 任何给定语句 生成的 自增量 都是 连续的，没有gap。 因为 table-level auto-inc lock 一直 保持到 语句结束， 并且 一次只能执行 一个 这样的语句。
   如果innodb_autoinc_lock_mode 设置为 2 (interleaved)。 "bulk insert" 生成的 自增

量 可能存在 gap ， 当且仅当 并发执行 "insert-like" 语句 时。
对于lock mode 1 或2， 连续语句 之间 可能存在 gap，因为对于 bulk insert ，可能不知道 每个语句 所需的 自增量的 确切数量，并且 可能会 高估。

Auto-increment values assigned by “mixed-mode inserts”
考虑 "mixed-mode inserts", 其中 "simple insert" 为某些(但不是全部) 结果行 指定自增量。这样的语句 在 lock mode 0,1,2， 中的 行为是不同的。
例如，假设 c1 是 表t1 的 auto_increment 列， 并且 最近自动生成的 序列号是 100。

```
mysql> CREATE TABLE t1 (
    -> c1 INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> c2 CHAR(1)
    -> ) ENGINE = INNODB;
```

考虑下面的 mixed-mode insert 语句：

```
mysql> INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'),
(NULL,'d');
```

在 innodb_autoinc_lock_mode 设置为0 (traditional)时， 有4个 new row:

```
mysql> SELECT c1, c2 FROM t1 ORDER BY c2;
+-----+------+
| c1  | c2   |
+-----+------+
|   1 | a    |
| 101 | b    |
|   5 | c    |
| 102 | d    |
+-----+------+
```

下一个可用的自增量 是103， 因为自增量一次分配一个，而不是在 语句执行 开始时 一次性分配。 无论 是否 并发 执行 "insert-like" 语句，这个结论 都正确。

lock mode=1。 有4个row

```
mysql> SELECT c1, c2 FROM t1 ORDER BY c2;
+-----+------+
| c1  | c2   |
+-----+------+
|   1 | a    |
| 101 | b    |
|   5 | c    |
| 102 | d    |
+-----+------+
```

但是，这种情况下， 下个可用 自增量 是 105 ，而不是103， 因为在处理 语句时， 分配了4个 自增量，但是 只使用了2个。 无论是否 同时执行 insert-like 语句，这个结论都正确。

lock mode=2, 4个新行

```
mysql> SELECT c1, c2 FROM t1 ORDER BY c2;
+-----+------+
| c1  | c2   |
+-----+------+
|   1 | a    |
```

```
|   x | b    |
|   5 | c    |
|   y | d    |
+-----+------+
```

x和y 是唯一值，并且 比之前生成的 任何行都大， 但是 x 和 y 的具体值 取决于 并发执行语句时 生成的 自增量的数量。

最后，考虑下列语句，最近生成的 自增量是100 ：

```
mysql> INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (101,'c'),
(NULL,'d');
```

所有 lock mode, 这个语句都会 导致 duplicate-key error 23000 （can’ t write; duplicate ket in table）。 因为 为(null,'b') 分配了 101， 所以 insert(101,'c') 失败。

Modifying AUTO_INCREMENT column values in the middle of a sequence of INSERT statements

mysql 5.7 及更早版本中， 修改 insert语句 序列 中间 的 auto_increment 列值 可能导致 duplicate key 错误。 例如，如果 你将 执行的 update 操作 将 auto_increment 列值 更改为 大于当前 最大 自增量的 值，则 后续的insert 可能遇到 duplicate key。

MySQL 8.0 及更早版本中，如果你将 auto_increment 列值 修改为 大于当前 最大自增量的值， 则 新值 被 持久化， 并且 后续 insert操作， 会从 新的 最大值 开始 分配自动增量值。

InnoDB AUTO_INCREMENT Counter Initialization

本节描述 innodb 如何初始化 auto_increment counter。

如果你为 InnoDB 表指定 auto_increment 列，则内存 表对象 包含 一个 称为 auto-increment counter 的 特殊 counter， 用于为 该列 分配 新值。

MySQL 5.7及更早之前， auto-increment counter 存储在 主内存上，而不是 磁盘上。需要在 服务器重启后 初始化 auto-increment counter， InnoDB 将在 第一次插入 包含 auto_increment 列的表 时 执行与 以下 语句等效的 语句：

```
SELECT MAX(ai_col) FROM table_name FOR UPDATE;
```

在MySQL 8.0中，这种行为 发生了 变化。 当前最大的 自动增量计数器 值 在每次 change 时 写入 redo log， 并保存到 每个 checkpoint 的 data dictionary 中。 这些 change 使 当前最大的 自增量计数器值 在服务器 重启时 保持不变。

在服务器 正常关闭后 重启时，InnoDB 使用 存储在 data dictionary 中的 当前最大 自增量 初始化 内存中的 自增量counter。

在 崩溃恢复 期间 服务器重启时， InnoDB 使用 存储在 data dictionary的 当前最大自动增量值 初始化 内存中的 自增量counter， 并 扫描 redo log 以查找 从上次 checkpoint 以来 写入的 自动增量计数器值。如果 redo log 中的 值大于 内存中的 自增量 counter 的值，则 使用 redo log 的值。 但是，在服务器意外退出的情况下，无法保证重用先前分配的 自增量。每次 由于 insert或 update 操作 更改了 当前最大自增量

时， 都会写入 redo log， 但 如果在 redo log 刷新到 磁盘 前 发生了 意外退出， 则 先前分配的 值 可能是 在 服务器 重启后 初始化 自增量counter 时重用。

InnoDB 使用等效的 select max(ai_col) from table_name for update 语句 来初始化 自增量counter 的 唯一情况是在 导入 没有 .cfg 元数据文件的 表时。否则，从 .cfg 元数据文件 中 读取当前 最大自增量计数器。 除了计数器值 初始化之外， 当尝试 将 计数器值 设置为 小于 。。。。 反正就是 通过 ALTER TABLE ... AUTO_INCREMENT = N 手动修改 自增量时 也会 select max(ai_col) from xxxxx 。 还有就是 要手动 改小的时候，需要确保 新的 自增量counter 大于 已有的 最大自增量值

在MySQL 5.7 和 更早版本中， 服务器 重启时， 会取消 auto_increment = N 这个 table option 的 效果，这个option 可以在 create table 和 alter table 语句中 分别用于 设置 初始counter 或 更改现有 counter值。
MySQL 8.0中，服务器重启 不会取消 这个 option 的效果。 如果 将 自增量计数器初始为 特定值，或者将 自增量counter 改为 更大的值，则 新值在服务器重启时保持不变。

ALTER TABLE ... AUTO_INCREMENT = N can only change the auto-increment counter value to a value larger than the current maximum.

MySQL 5.7 及更早版本中， 在 RollBack 操作 之后 立刻 重启服务器 可能导致 重用 之前分配给 回滚事务的 自增量，从而 有效地回滚 当前最大的 自增量。
MySQL 8.0中，当前最大的 自增量 被持久化， 防止重复使用 以前分配的值。

如果 show table status 语句 在 自增量counter 初始化之前 检查表，innodb 会打开表 并使用 存储在 data dictionary 中的 当前 最大 自增量 初始化 counter值。 然后将 该值 存储在 内存中 以供 以后的 insert 或 update 使用。 counter值的 初始化 在表上 使用了 普通的 排它锁读取， 直到 事务结束。 InnoDB 在 为 用户指定自增量大于0 的新表 的初始化 自增量 counter 时 遵循 相同的过程。

自增量计数器 初始化后， 如果 insert row时 没有 显式指定 自增值， InnoDB 会 隐式递增计数器 并将 新值分配给 该列。 如果 insert 的row 有显式 的 自增值，并且 该值 大于 当前 最大 counter值， 则 将 计数器 值 设置为 显式的值。

只要服务器运行，InnoDB 就会 使用 内存中 的 自增量计数器，当 服务器停止并 重启时，InnoDB 重新初始化 自增量计数器。

auto_increment_offset 变量 决定了 auto_increment 列值的 起点。默认 1。
auto_increment_increment 控制了 连续 列值 之间的 间隔， 默认为 1.

Note

当Auto_increment 溢出时， 后续的 insert 返回 duplicate-key error。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-indexes.html
15.6.2 Indexes

15.6.2.1 Clustered and Secondary Indexes

15.6.2.2 The Physical Structure of an InnoDB Index
15.6.2.3 Sorted Index Builds
15.6.2.4 InnoDB Full-Text Indexes

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html
15.6.2.1 Clustered and Secondary Indexes

每个InnoDB表 都有一个 特殊的index， 称为 clustered index,它存储了 row data。
通常，聚簇index和 主键 同义。
为了从 query，insert，和 其他 数据库操作 中 获得 最佳性能， 了解 InnoDB 如何使用 聚簇index 来优化 常见的 查找 和 DML 操作 非常重要。

1. 当你在表上 定义 primary key时， InnoDB 将其 作为 聚集index。 应该为每个表 定义一个 主键。 如果 没有 逻辑唯一 且 非空的 列 或 列集 来 作为主键，请添加一个 自增量列。 自增量列 是唯一的，并在 插入 新row 时 自动添加。
2. 如果你没有为 表 定义 主键，则 InnoDB 使用 第一个 所有key column 都定义为 not null 的 unique index 作为 clustered index。
3. 如果表没有 primary key 或 合适的 unique index， InnoDB 会在 包含 row id 值的 合成列(synthetic column) 上 生成一个 名为 GEN_CLUST_INDEX 的 隐藏聚集index。 row id 是一个 6字节的字段，随着新行的插入 而 单调递增。所以，按row id排序后的行，在物理上是 按照 这个顺序 insert 的。

How the Clustered Index Speeds Up Queries
通过 聚集index 访问 row 很快，因为 index search 直接 指向了 包含 row data 的 page。 如果 表很大， 聚集index 经常节约 磁盘IO， 和 那些 不是保存在 index 记录的 page 上的 存储结构相比。

How Secondary Indexes Relate to the Clustered Index
聚集index之外的 index 被称为 二级index。在InnoDB中，二级index中的 每条记录 都包含 row 的 主键列，以及 为 二级index 指定的 列。 InnoDB 使用这个 主键值 来 搜索 聚集index 中的 row

如果主键长， 二级index 占用的 空间 就更多， 所以 主键短 是有利的。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html

15.6.2.2 The Physical Structure of an InnoDB Index

除了 spatial index (空间index)， InnoDB index 都是 B tree 数据结构。 空间index 使用 R 树，它是 用于 索引 多维数据的 专用 数据结构。

index record 存储在 B 树 或 R树 的 leaf page 中。 index page 的默认大小是 16kb。 这个大小 是通过 innodb_page_size 配置的。

。。聚集index 是直接把 row 放在 page 中，那么 如果 row 大于16k， 是什么情况？

当新record 插入到 聚集index 中时， innodb 会尝试 留出 1/16 的 page空间 以供 将来的 插入 和 更新 。 如果 index record 是 按序(升序或降序) 插入，则 生成的 index page 大约是 15/16 满， 如果 是 随机插入 record， 那么 是 1/2 到 15/16 满。

InnoDB 在创建 和 重建 B树 index 时 执行 批量加载。 这种 创建index 的方法 被称为 sorted index build。 innodb_fill_factor 变量 定义了 在 sorted index build 期间 每个 B树 page 的 填充的空间百分比，剩余空间 保留 用于 未来index 增长。
空间index 不支持 sorted index build。
将 innodb_fill_factor 设置为100 后， 聚集index page 中 1/16 的空间 用于 未来的 index 增长。

如果InnoDB index page 的 fill因子 小于 MERGE_THRESHOLD (如果没有定义，默认50%)，InnoDB 尝试 收缩 index tree 以释放 page。 MERGE_THRESHOLD 适用于 B树 和 R树。

。。那就是 一个page 有多个 index ？ 不然 怎么收缩 释放。。

---

https://dev.mysql.com/doc/refman/8.0/en/sorted-index-builds.html

15.6.2.3 Sorted Index Builds

innodb 在 create 或 rebuild index时， 执行bulk load, 而不是 一次 插入一条index。 这种 索引创建的 方法 也被称为 sorted index build。sorted index build 不支持 spatial index。

index build 分为 3个阶段：

1. 扫描聚集index，生成 index entry，并且放到 sort buffer。 当 sort buffer 变满，entry 被排序，并写入 临时文件。 此过程 也被称为 "run"
2. with 一个或多个 run 写入的 临时文件， 对文件中的 所有 entry 执行 merge sort 。
3. 排序后的 entry 被插入到 B树， 从MySQl 8.0.31 开始，这个阶段是 多线程的。

在 使用 sorted index build 之前 是 使用 insert API 来将 index entry 一次插入一条 到B树。 这个方法涉及 打开 B 树 游标 以查找 插入位置，然后 使用 optimistic insert 来将 entry 插入到 B树 page。 如果 由于 page 满 而 insert失败，会执行 pessimistic insert (悲观插入)， 这涉及 打开 B树 游标 并根据需要 拆分 和 合并 B树 节点 来寻找 可用的空间 for entry。 这种 "top-down" 方式构建的索引 的 缺点是 搜索插入位置的cost 和 B树节点的 不断地 拆分 和 合并的 cost。

sorted index build 使用 "bottom up" 方式来构建 index。 使用这种方法，(MySQL/InnoDB) 维护了 B树 所有 层级的 最右 leaf page 的 指针， 必要时 新建一个 最右 leaf page。 按照 排序后顺序 插入 entry。一旦 leaf page 满，添加一个 node pointer 到 parent page，并为 下一次insert 分配一个 兄弟 leaf page。 这个 过程 持续 直到 所有条目 被插入，这

可能导致 插入 insert up to the root level。分配 兄弟 page 时，释放对 之前固定的 leaf page 的 引用，新分配的 leaf page 变成 最右 leaf page 和新的默认 insert 位置。

Reserving B-tree Page Space for Future Index Growth

要为 未来的 index growth 留出空间， 你可以使用 innodb_fill_factor 来保留 一定 百分比的 B树 page 空间。 例如 设置 innodb_fill_factor 为80，则在 sorted index build 期间 保留 B树 page 的 20% 的空间。 这个 设置 适用于 B树的 lesf 和 non-lesf page。 不适用于 用于text 或 blob entry 的 external page。 保留的空间 可能和 配置 不完全相同， 因为 innodb_fill_factor 是 hint 而不是 hard limit。

Sorted Index Builds and Full-Text Index Support
sorted index build 支持 fulltext index。以前，使用sql 来插入 entry 到 fulltext index。

Sorted Index Builds and Compressed Tables
对于压缩表，以前的 index 创建方法 将 entry 追加到 压缩page 和 非压缩 page。 当 modification log(表示压缩page上额 可用空间) 变满时， 将重新压缩 压缩page。 如果由于空间不足 而导致 压缩失败，page 会被 拆分。
通过 sorted index build,entry 只会被 append 到 未压缩page。当 未压缩page 变满，它被压缩。 自适应填充(adaptive padding) 用于确保 大多数情况下 压缩成功，但如果压缩失败，则会拆分page 然后再次压缩。 直到 压缩成功。

Sorted Index Builds and Redo Logging
在 sorted index build 期间 redo logging 被禁用。有一个 checkpoint 来确保 index build 可以 承受意外退出 或失败。 checkpoint 强制 将所有 dirty page 写入到 磁盘。在 sorted index build 期间， page cleaner 线程 会定期 收到信号 来 flush dirty page，确保可以快速处理 checkpoint 操作。 通常，当 clean page 的数量 低于 设置的阈值时，page cleaner 线程会 flush dirty page。 对于 sorted index build， dirty page 会被 及时 flush 以减少 checkpoint 的 并行IO 和 CPU 的开销。

Sorted Index Builds and Optimizer Statistics
sorted index build 可能导致 优化器 统计信息 与 以前的索引创建方法 生成的 统计信息不同。 统计信息的不同，并不影响 工作负载性能， 因为 这个不同 是由于 填充index 的 算法不同导致的。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-fulltext-index.html

15.6.2.4 InnoDB Full-Text Indexes

全文index 是 在基于文本的列 上创建的， 如 char，varchar，text。 以加快对这些列 中数据

的 查询 和 DML 操作。

全文index 被定义为 create table 的一部分，或 通过 alter table 或 create index 来添加到 现有表 中。

全文搜索 使用 match() .. against 语法

InnoDB Full-Text Index Design

innodb 全文索引 采用 inverted index design。 inverted index 保存了 word 的list，对于每个 word，(保存) 了 这个word 出现的 document 的list。 为了支持 邻近搜索 (proximity search)， 还存储了 每个单词的 位置信息，作为 byte offset。

InnoDB Full-Text Index Tables
在 创建 innodb 全文index 时，会创建 一组 index table。如下所示：

```
mysql> CREATE TABLE opening_lines (
       id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200),
       FULLTEXT idx (opening_line)
       ) ENGINE=InnoDB;

mysql> SELECT table_id, name, space from INFORMATION_SCHEMA.INNODB_TABLES
       WHERE name LIKE 'test/%';
+----------+--------------------------------------------------------+-------+
| table_id | name                                                   | space |
+----------+--------------------------------------------------------+-------+
|      333 | test/fts_0000000000000147_00000000000001c9_index_1 |   289 |
|      334 | test/fts_0000000000000147_00000000000001c9_index_2 |   290 |
|      335 | test/fts_0000000000000147_00000000000001c9_index_3 |   291 |
|      336 | test/fts_0000000000000147_00000000000001c9_index_4 |   292 |
|      337 | test/fts_0000000000000147_00000000000001c9_index_5 |   293 |
|      338 | test/fts_0000000000000147_00000000000001c9_index_6 |   294 |
|      330 | test/fts_0000000000000147_being_deleted            |   286 |
|      331 | test/fts_0000000000000147_being_deleted_cache      |   287 |
|      332 | test/fts_0000000000000147_config                   |   288 |
|      328 | test/fts_0000000000000147_deleted                  |   284 |
|      329 | test/fts_0000000000000147_deleted_cache            |   285 |
|      327 | test/opening_lines                                 |   283 |
+----------+--------------------------------------------------------+-------+
```

前6个index table包含 inverted index，称为 辅助索引表 (auxiliary index tables)。当传入的文档 被 标记时 (tokenized)，单个word (也被称为 token) 连同位置信息 和 关联的 DOC_ID 一起插入到 index table 中。 word 被全部排序， 然后根据 第一个char 的 权重 分配到 6个 index table 之一 来分组。

inverted index 被 划分为 6个 辅助索引表，以支持 parallel index creation。 默认下，2个线程 对 word 和 相关数据 进行 标记，排序，插入 到index table。 线程数 通过 innodb_ft_sort_pll_degree 来控制。 在大型表 上 创建 全文index时， 请考虑增加 线程数。

辅助索引表 表名 以 fts_ 为前缀， 以 index_# 为后缀。 每个辅助index表 和 indexed 表相关联，通过 辅助索引表 表名中 与 indexed table 的 table_id 匹配的 16进制值。 例如 test/opening_lines 表的 table_id 为 327， 16进制是 0x147. 147就出现了 与 test/opening_lines 表 关联的 辅助索引表 的名称中。

表示 全文索引的 index_id 的 16进制值 也出现在 辅助索引表 表名中。例如，辅助表名 test/fts_0000000000000147_00000000000001c9_index_1 中，0x1c9 的十进制 是 457。 在 opening_lines 表 (idx) 上定义的 索引 可以通过 查询 information_schema.innodb_indexes 表 中 该值(457)来识别。

```
mysql> SELECT index_id, name, table_id, space from
INFORMATION_SCHEMA.INNODB_INDEXES
       WHERE index_id=457;
+----------+------+----------+-------+
| index_id | name | table_id | space |
+----------+------+----------+-------+
|      457 | idx  |      327 |   283 |
+----------+------+----------+-------+
```

如果在 file-per-table 表空间中 创建主表，则 index tables 保存在它们 自己的 表空间中。 否则，index tables 将存储在 indexed table(被索引的表) 所在的 表空间中。

上例中，显示的 其他index table 被称为 common index table， 用于 删除 和 存储 全文 index 的 内部状态。 与为每个 全文index 创建的 inverted index table 不同， 这组表 对于在 特定表 上 创建 的 所有 全文index 都是通用的。

即使 全文index 被删除， common index table 也会被保留。删除 全文索引时， 会保留 为该index 创建的 FTS_DOC_ID 列，因为 删除 FTS_DOC_ID 列 将 需要 重建 先前 indexed table。 common index tables 用来 管理 FTS_DOC_ID 列。

fts_*_deleted and fts_*_deleted_cache

包含已删除但 数据还没有 全文index 中删除的 document id (DOC_ID)。
fts_*_deleted_cache 是 fts_*_deleted 表的内存版本

fts_*_being_deleted and fts_*_being_deleted_cache

包含 已删除，且 当前正在 从 全文index 中 删除 其数据的 文档 的 文档ID (DOC_ID).
_cache是内存版本

fts_*_config

存储有关 全文index 的 内部状态的 信息。最重要的是， 它存储 FTS_SYNCED_DOC_ID，这个 ID 标识 已解析 并 flush 到磁盘的 文档。 在崩溃恢复的情况下，FTS_SYNCED_DOC_ID 值用于 标识 尚未刷新到 磁盘的文档，以便可以 重新解析 文档 并将其 添加会 全文索引 缓存。要查看此表的数据，query information_schema.innodb_ft_config 表。

InnoDB Full-Text Index Cache
插入文档时，对其将进行标记，并将单个word 和 相关数据 插入到 全文index 中。这个过程中，即使对于 小文档， 也可能导致 对 辅助index表的 大量 小insert，从而使得 对这些表的 并发访问 成为 竞争点。为了避免这个问题，innodb 使用 full-text index cache 来 临时缓存 最近插入的 行的 index table insertions。 这个 in-memory cache 维持 insertion，直到 cache 满， 然后 batch flush 到 disk (到 辅助索引表)。 你可以query information_schema.innodb_ft_index_cache 查看 最近insert的row 的 tokenized data。

caching 和 batch flushing 行为 避免了 对 辅助index表的 频繁 更新，这个可能导致 在 繁忙的 insert 和 update 中 出现 并发访问问题。 批处理技术 还避免了 同一个 word 的 多次 insert，并 最大程度 减少了 重复 条目。 不是 单独flush 每个 word，而是 将 相同 word 的 insert 合并， flush 一个entry 到 disk， 提高了 插入效率，同时 保持 辅助索引表 尽可能小。

innodb_ft_cache_size 用于 配置 全文index cache size (基于每个表)， 这项配置 会 影响 全文index cache 刷新的 频率。 你还可以使用 innodb_ft_total_cache_size 来为 给定实例中的 所有表 定义 全局 全文index cache 大小限制。
。。total cache size 是 整个MySQL 只能用 这么多内存， 还是说 每个 表 只能用这么多？

全文index cache 存储了 和 辅助index表 相同的 信息。 但是，全文index cache 只 cache 最近insert的row 的 tokenized data。 已经flush 到disk(到 辅助idnex表) 的 数据 不会在query时 再带回到 cache中。 辅助index表中的数据 是直接被查询的 (。。不走cache)，将辅助index 表中的 结果 和 全文index cache 中的 结果 合并后 返回。

InnoDB Full-Text Index DOC_ID and FTS_DOC_ID Column
InnoDB 使用称为 DOC_ID 的唯一文档 标识符 将 全文index 中的 word 映射到 word出现的 document。 这个mapping 需要 被index表的 FTS_DOC_ID 列。 如果没有FTS_DOC_ID，InnoDB 会在 创建 全文index 时 添加 一个 隐藏的 FTS_DOC_ID 列。下面的例子 演示了这种行为。

下表不包括 FTS_DOC_ID 列

```
mysql> CREATE TABLE opening_lines (
       id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200)
       ) ENGINE=InnoDB;
```

当你使用 create fulltext index 语法在 表上 创建 全文index时， 会返回一个 warning，报告 InnoDB 正在 rebuild 表 以添加 FTS_DOC_ID 列。

```
mysql> CREATE FULLTEXT INDEX idx ON opening_lines(opening_line);
Query OK, 0 rows affected, 1 warning (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------------+
| Level   | Code | Message                                             |
```

```
+---------+------+-----------------------------------------------+
| Warning |  124 | InnoDB rebuilding table to add column FTS_DOC_ID |
+---------+------+-----------------------------------------------+
```

当使用 alter table 来 添加 全文index 到 没有FTS_DOC_ID列的 table 时 也会 有 相同的警告。 如果你在 create table 时 创建了 全文index， 但没有 声明 FTS_DOC_ID 列，innodb 会自动增加一个 hidden FTS_DOC_ID 列，不会有 warning

在 create table 时 定义 FTS_DOC_ID 列 比 在已存在数据的 表上 创建 全文index 的 成本更低。

如果不关心 create fulltext index 的性能，请忽略 FTS_DOC_ID 列，让innodb 创建。
。。。这个2条sql 就可以了啊， 一条 create table， 第二条 create fulltext index 就可以了啊， 空表的消耗应该是0。
如果让innodb创建 FTS_DOC_ID，行为： InnoDB creates a hidden FTS_DOC_ID column along with a unique index (FTS_DOC_ID_INDEX) on the FTS_DOC_ID column.
。。也不是，这里 还 创建了 唯一index 。(结合下面的) 所以 自己手工创建的话 可以 不建立 index， auto_increment 应该够了， 只要自己不 手动插入数据。。
。。不，下面说了，如果没有 这个索引，innodb 会创建。。

如果要创建 自己的 FTS_DOC_ID，必须定义为 bigint unsigned not null,并命名为 全大写的FTS_DOC_ID。

```
mysql> CREATE TABLE opening_lines (
       FTS_DOC_ID BIGINT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200)
       ) ENGINE=InnoDB;
```

。。？但是这里 没有在 创建时 声明 全文index （虽然不知道能不能。。）。。那么 还需要一条 来 创建 full-text index， 那么 自己手动 创建这个列，没有什么意义。。 也不是。。 这里是 主键。 就是 它把 一个 系统列 加上了 自己的 含义。 可以作为 主键。

如果你选择 自己定义 FTS_DOC_ID 列，则你有责任 管理 该列，避免 空值或重复值。
或者，你可以在 FTS_DOC_ID 列上 创建唯一 FTS_DOC_ID_INDEX 索引。

```
mysql> CREATE UNIQUE INDEX FTS_DOC_ID_INDEX on opening_lines(FTS_DOC_ID);
```

If you do not create the FTS_DOC_ID_INDEX, InnoDB creates it automatically.
。。。。

FTS_DOC_ID_INDEX 不能定义为 降序index，因为 innodb sql解析器 不使用 降序index。
已使用的 FTS_DOC_ID 和 新的FTS_DOC_ID 之间的 最大 gap 是 65535
为了避免rebuild table， 删除 全文index时， FTS_DOC_ID 列被保留。

InnoDB Full-Text Index Deletion Handling

删除具有 全文index 列 的 record 可能导致 辅助index表中 大量 小删除，从而 使得 对这些表的 并发访问 成为了 竞争点。
为了避免这个问题，每当从 indexed 表中 删除记录时，被删除的 文档的 DOC_ID 被 记录到

特殊的 FTS_*_DELETED 表中，并且 indexed record 被保留在 全文index 中。在返回查询结果之前，使用 FTS_*_DELTETED 中的信息 来过滤掉 已经被删除的 DOC_ID。
这个设计的好处是 快 且 成本低。 缺点是 删除记录后 index 的大小不会立即减小。
要删除 已删除记录的 全文index entry， 请在 innodb_optimize_fulltext_only=ON 的 被index表 上 执行 optimize table 来 重建 全文index。
。。？这个是 不会立即减小，还是 永远不会减小 除非 optmize table ？ 感觉是 永远不会减小。 这样的话 FTS_*_DELETED 表 越来越大。 也会 瓶颈，， 所以 应该是 定期 optimzie table。 。 看这个表 删除 频不频繁了。

InnoDB Full-Text Index Transaction Handling
innodb 全文index 由 特殊的事务处理特性， 由于 它的 caching 和 batch procesing。具体来说，全文index 的 update 和insert 在 事务 commit 时 处理， 这意味着 全文index 只能看到 committed 数据。
下面的例子 演示了这种行为，全文search 只 返回 commit 后的数据。

```
mysql> CREATE TABLE opening_lines (
       id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
       opening_line TEXT(500),
       author VARCHAR(200),
       title VARCHAR(200),
       FULLTEXT idx (opening_line)
       ) ENGINE=InnoDB;

mysql> BEGIN;

mysql> INSERT INTO opening_lines(opening_line,author,title) VALUES
       ('Call me Ishmael.','Herman Melville','Moby-Dick'),
       ('A screaming comes across the sky.','Thomas Pynchon','Gravity\'s
Rainbow'),
       ('I am an invisible man.','Ralph Ellison','Invisible Man'),
       ('Where now? Who now? When now?','Samuel Beckett','The Unnamable'),
       ('It was love at first sight.','Joseph Heller','Catch-22'),
       ('All this happened, more or less.','Kurt Vonnegut','Slaughterhouse-
Five'),
       ('Mrs. Dalloway said she would buy the flowers herself.','Virginia
Woolf','Mrs. Dalloway'),
       ('It was a pleasure to burn.','Ray Bradbury','Fahrenheit 451');

mysql> SELECT COUNT(*) FROM opening_lines WHERE MATCH(opening_line)
AGAINST('Ishmael');
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+

mysql> COMMIT;

mysql> SELECT COUNT(*) FROM opening_lines WHERE MATCH(opening_line)
```

```
AGAINST('Ishmael');
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
```

Monitoring InnoDB Full-Text Indexes
你可以监控和检查 innodb 全文 index 的 特殊的 文本处理 ，通过 query information_schema 中的表

- INNODB_FT_CONFIG
- INNODB_FT_INDEX_TABLE
- INNODB_FT_INDEX_CACHE
- INNODB_FT_DEFAULT_STOPWORD
- INNODB_FT_DELETED
- INNODB_FT_BEING_DELETED

你还可以通过 查询 innodb_indexes 和 innodb_tables 查看 全文 index 和 表的 基本信息。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-tablespace.html

15.6.3 Tablespaces

15.6.3.1 The System Tablespace
15.6.3.2 File-Per-Table Tablespaces
15.6.3.3 General Tablespaces
15.6.3.4 Undo Tablespaces
15.6.3.5 Temporary Tablespaces
15.6.3.6 Moving Tablespace Files While the Server is Offline
15.6.3.7 Disabling Tablespace Path Validation
15.6.3.8 Optimizing Tablespace Space Allocation on Linux
15.6.3.9 Tablespace AUTOEXTEND_SIZE Configuration

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-system-tablespace.html
15.6.3.1 The System Tablespace

system tablespaces 是 change buffer 的 存储区域。 也包含 那些 创建在 system tablessapces (而不是 file-per-table or general tablesapces) 的表 和 索引数据。
以前版本，sysmte tablespaces 包含 innodb data dictionary。MySQL 8.0，innodb 存储 metadata 到 MySQL data dictionary.
以前版本，system tablespaces 包含 doublewrite buffer storage area，8.0.20开始，这个存储区域 位于 单独的 doublewrite files。

system tablespaces 有 一个或多个 data 文件， 默认下， 一个单独的 system tablespaces data file， 名为 ibdata1 ，创建在 data directory 中。 system tablesspaces data file 的 大小 和 数量 由 innodb_data_file_path 启动项 定义。

Resizing the System Tablespace

Increasing the Size of the System Tablespace

The easiest way to increase the size of the system tablespace is to configure it to be auto-extending. To do so, specify the autoextend attribute for the last data file in the innodb_data_file_path setting, and restart the server. For example:

innodb_data_file_path=ibdata1:10M:autoextend

When the autoextend attribute is specified, the data file automatically increases in size by 8MB increments as space is required. The innodb_autoextend_increment variable controls the increment size.

You can also increase system tablespace size by adding another data file. To do so:

1. stop mysql server
2. 如果 innodb_data_file_path 定义的最后一个 data file 是 autoextend，请将其删除，并修改size 属性 反应当前数据的大小。这个大小 是 检查 文件系统的 文件大小， 然后将其 向下 舍入 到 最近的 MB值， 1MB = 1024*1024 bytes,
3. 将 新data file append 到 innodb_data_file_path 配置，可以指定为 autoextend。 只能为 innodb_data_file_path 的 最后一个 data file 指定 autoextend。
4. 启动mysql server

例如，这个tablespace 有 一个 auto-extending 的 data file:

```
innodb_data_home_dir =
innodb_data_file_path = /ibdata/ibdata1:10M:autoextend
```

假设 data file 是 988mb。那么 新配置：

```
innodb_data_home_dir =
innodb_data_file_path = /ibdata/ibdata1:988M;/disk2/ibdata2:50M:autoextend
```

添加 新的 data file 时， 不要 指定 已存在的名字，server启动时， innodb 会根据 配置，新建 并初始化 不存在的 文件。

You cannot increase the size of an existing system tablespace data file by changing its size attribute. For example, changing the innodb_data_file_path setting from ibdata1:10M:autoextend to ibdata1:12M:autoextend produces the following error when starting the server:

```
[ERROR] [MY-012263] [InnoDB] The Auto-extending innodb_system
data file './ibdata1' is of a different size 640 pages (rounded down to MB)
than
specified in the .cnf file: initial 768 pages, max 0 (relevant if non-zero)
```

pages!

Decreasing the Size of the InnoDB System Tablespace
Decreasing the size of an existing system tablespace is not supported. The only option to achieve a smaller system tablespace is to restore your data from a backup to a new MySQL instance created with the desired system tablespace size configuration.

要避免使用 large system tablespaces， 考虑使用 file-per-table 或 general 表空间 保存你的 data。 file-per-table 是默认表空间类型。 和 system 表空间不同，file-per-table 表空间在被截断 或删除时 会将 磁盘空间返回给 OS。

Using Raw Disk Partitions for the System Tablespace
raw disk partitions 可以作为 系统表空间的 数据文件。 这个技术 可以在 window 和 一些 linux/unis 上 实现 非缓冲IO (nonbuffered IO)， 而无需 文件系统的开销。 对 使用 和 不使用 原始分区 进行测试，来验证 是否提高 性能。

当使用 原始磁盘分区时，请确保 运行MySQL 服务器的 user id 对 该分区 有 读写权限。例如 如果使用mysql 用户 运行 server， 那么 mysql 用户 必须能对 分区进行 读写。 如果使用 --memlock 选项 运行 server，那么 必须用 root 用户运行，所以 分区 必须可以被 root 进行 读写。

Allocating a Raw Disk Partition on Linux and Unix Systems

Allocating a Raw Disk Partition on Windows

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-file-per-table-tablespaces.html
15.6.3.2 File-Per-Table Tablespaces

file-per-table 表空间 包含 单个innodb表的数据和index， 并存储在 文件系统中的 单个数据文件中。

File-Per-Table Tablespace Configuration
默认下，innodb 在 file-per-table 表空间 创建 表。这个通过 innodb_file_per_table 控制，禁用它，会导致 innodb 在 system 表空间 创建 表。

可以在 option file中 配置 innodb_file_per_table， 也可以在 运行时 使用 set global 进行配置，这个需要足够的权限。
[mysqld]

```
innodb_file_per_table=ON

mysql> SET GLOBAL innodb_file_per_table=ON;
```

File-Per-Table Tablespace Data Files

file-per-table 表空间 被创建在 mysql data directory 下的 schema directory 的 一个 .ibd 数据文件中。 .idb文件 以 表命名。

你可以使用 create table 语句的 data directory 子句 在 数据目录之外 隐式创建一个 file-per-table 表空间数据文件。

File-Per-Table Tablespace Advantages

file-per-table 和 shared tablespaces (如 system tablespace, general tablespaces) 相比的 优势

1. 在截断 或 删除 在 file-per-table 表空间 中创建的 表后， 磁盘空间 将会被还给 OS。 截断或删除 存储 在 共享表空间 中表 会在 共享表空间 数据文件中 创建 可用空间，该空间 只能用于 innodb数据， 不会还给OS，所以 文件不会变小。
2. 对 shared 表空间中的 表 执行 table-copying alter table 操作会增加 表空间占用的磁盘空间。此类操作 可能需要 与 表+索引 一样多的 额外空间。该空间不会 像 file-per-table 表空间那样 释放回 OS。
3. truncate table 在 file-per-table 表空间的 表上 执行时 性能更好。 。。。截断表，是清空表数据，和delete 相比 ， 性能更高，不触发 触发器，删除全部(没有where子句)，不能回滚。
4. file-per-table 表空间 data file 能 在 单独的 存储设备上 创建， for 优化IO，空间管理，备份。
5. 你可以从 另一个MySQL 实例 导入 file-per-table 表空间 中的表。
6. 在file-per-table 表空间 创建的表 支持 dynamic 和 compressed row format，这2个在 system tablespace 不被支持。
7. 当数据损坏，备份，或二进制日志不可用或 MySQL无法启动时， 存储在 单个 表空间 数据文件中的 表可以节约时间 并提高 成功恢复的机会。
8. file-per-table 表空间中创建的 表 可以使用 MySQL Enterprise Backup 快速备份或恢复，而不会中断 其他 innodb表的 使用。
9. file-per-table 表空间 允许 通过 监视 表空间数据文件 的大小 来 监视 表的大小。
10. 当 innodb_flush_method 设置为 O_DIRECT 时，常见的 linux 文件系统 不允许 并发写入单个文件，因此 将 file-per-table 和 这个配置 结合，可能会提高性能。
11. 共享表空间 中的表 受 64TB 表空间大小限制，相比之下，每个file-per-table 表空间的大小 限制为 64TB。 。。应该是 文件系统最大的单个文件是64TB？要具体看情况吧。

File-Per-Table Tablespace Disadvantages

和共享表空间对比， file-per-table有以下 劣势：

1. 使用file-per-table 表空间，每个表都可能有 未使用的空间，这些空间只能由 一个表的 row 使用，如果管理不当，会造成空间浪费。
2. fsync 操作 在 多个 file-per-table 数据文件 而不是 单个 共享表空间 数据文件上 执行。由于 fsync 是针对 每个文件的，因此 无法组合 多个表的 写入操作。这可能导致 fsync 操作的 总数增加。
3. mysqld 必须为 每个 file-per-table 表空间 保留 一个 打开的 文件句柄，如果有许多表，可能影响性能。

4. 当每个表 都有 自己的 数据文件时，需要更多的 file descriptor。
5. 可能会出现更多的 碎片，这会阻碍 drop table 和 table scan 性能。但是，如果这些碎片 被管理，file-per-table 表空间 可以提高这些操作的性能。
6. drop file-per-table的table 时 会 扫描 buffer pool。这个扫描 对于 大型buffer pool 可能要持续 几秒。 scan时 使用了 board internal lock，这会 delay 其他操作。
7. innodb_autoextend_increment 变量 定义了 在自动扩展 共享表空间 变满时 扩展其 大小的增量， 不适用于 file-per-table 表空间。 file-per-table 中表的扩展 最初 是少量的， 之后 扩展 以 4mb 为 增量发生。

---

https://dev.mysql.com/doc/refman/8.0/en/general-tablespaces.html

15.6.3.3 General Tablespaces

general 表空间 是使用 create tablespace 创建的 共享innodb 表空间。

General Tablespace Capabilities
有以下能力：

1. 与 system表空间 类似，general 表空间 是 能 存储多个表的 表空间。
2. 和 file-per-table 表空间相比， general 具有 潜在的 内存优势。 服务器在 表空间的整个生命周期内 将表空间元数据 保存在内存中。 和 file-per-table相比，都有 相同数量的表 时，general 消耗的 元数据内存 更少。
3. general 表空间 数据文件 可以放置 在 相对于 或 独立于 MySQL 数据目录的 目录中，这为你 提供了 file-per-table 表空间的 多数据文件 和 存储管理 的能力。 与file-per-table 表空间一样，将 数据文件 放置在 MySQL 数据目录 之外的 能力 允许你 单独 管理 关键表的 性能，为 特定表 设置 RAID 或DRBD，或绑定到 特定磁盘。

。。DRBD 是由内核模块和相关脚本而构成，用以构建高可用性的集群。其实现方式是通过网络来镜像整个设备。您可以把它看作是一种网络RAID

4. general 表空间 支持 所有的 row farmat 和 相关功能。
5. tablespace 选项 可以和 create table 一起使用，以在 general 表空间，file-per-table 表空间 或 system 表空间 中 创建表。
6. tablespace 选项 可以和 alter table 一起使用 来 在 general,file-per-table，system 表空间 之间 移动表。

Creating a General Tablespace
General tablespaces are created using CREATE TABLESPACE syntax.

```
CREATE TABLESPACE tablespace_name
    [ADD DATAFILE 'file_name']
    [FILE_BLOCK_SIZE = value]
        [ENGINE [=] engine_name]
```

可以在 data directory 中 或 之外 创建 general 表空间。为了 避免 与 隐式创建的 file-per-table 表空间 冲突，不支持 在 data directory 下的 子目录中 创建 通用表空间。

在 data directory 之外 创建 通用表空间时， 该目录 必须 存在并且 必须在 创建表之前 被 innodb 所知道。 要让 innodb 知道 未知目录，请将 目录 添加到 innodb_directories 参数，这个参数是 只读参数，配置后需要重启服务器。

例子，在 data directory 中创建 通用表空间：

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE 'ts1.ibd' Engine=InnoDB;
```

or

```
mysql> CREATE TABLESPACE `ts1` Engine=InnoDB;
```

从8.0.14 开始，add datafile 子句 是可选的， 之前是必须的。如果 创建表空间时 没有 指定 add datafile 子句，则会隐式创建 具有 唯一 文件名的 表空间 数据文件。 这个唯一文件名是 128位uuid， 格式为 5组 由 破折号 分隔 的 16进制数字 (aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee)。 通用表空间 数据文件 包括了 .ibd 后缀。 在 replication 环境中，在 源上 创建的 数据文件名 和 副本上 创建的 数据文件名 不同。

在 data directory 外 创建 general 表空间

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE '/my/tablespace/directory/ts1.ibd'
Engine=InnoDB;
```

只要 表空间 目录 不再 data directory 下， 你就可以 指定 相对于 data directory 的路径，下面的例子中， my_tablespace 目录 和 data directory 处于同一级别：

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE '../my_tablespace/ts1.ibd'
Engine=InnoDB;
```

create tablespace 中 必须 定义 engine=InnoDB， 或 InnoDB 是默认 存储引擎。

Adding Tables to a General Tablespace

create table tbl_name .. tablespace [=] tablespace_name

or

alter table tbl_name tablespace [=] tablespace_name

```
mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1;

mysql> ALTER TABLE t2 TABLESPACE ts1;
```

MySQL 5.7.24 中不推荐 将 表分区 添加到 共享表空间，并在 MySQL 8.0.13 中删除。 共享表空间 包括 InnoDB 系统表空间 和 通用 表空间。

General Tablespace Row Format Support

通用表空间 支持 所有的 row format （ REDUNDANT、COMPACT、DYNAMIC、COMPRESSED ），但需要注意的是，由于 物理page size 的不同，压缩表 和 为压缩表 不能 在同一个 通用表空间 中 共存。

对于 包含 压缩表 (row_format = compressed) 的 general 表空间， 必须指定 file_block_size 选项，并且 值 必须是 与 innodb_page_size的值 相关的 有效压缩页面大

小。 此外，压缩表的 物理页 大小 (KEY_BLOCK_SIZE) 必须等于 FILE_BLOCK_SIZE/1024。 例如，如果 innodb_page_size=16kb 且 FILE_BLOCK_SIZE=8， 那么 表的 KEY_BLOCK_SIZE 必须是 8。

下标显示了 允许的 innodb_page_size， FILE_BLOCK_SIZE， KEY_BLOCK_SIZE 组合。 FILE_BLOCK_SIZE 值也可以 以字节 为 单位 指定。 要确定 给定 FILE_BLOCK_SIZE 的 有效 KEY_BLOCK_SIZE 值， 请将 FILE_BLOCK_SIZE 值除以 1024. 表压缩 不支持32k 和 64k innodb page 大小。

InnoDB Page Size (innodb_page_size)	Permitted FILE_BLOCK_SIZE Value	Permitted KEY_BLOCK_SIZE Value
64KB	64K (65536)	Compression is not supported
32KB	32K (32768)	Compression is not supported
16KB	16K (16384)	None. If innodb_page_size is equal to FILE_BLOCK_SIZE, the tablespace cannot contain a compressed table.
16KB	8K (8192)	8
16KB	4K (4096)	4
16KB	2K (2048)	2
16KB	1K (1024)	1
8KB	8K (8192)	None. If innodb_page_size is equal to FILE_BLOCK_SIZE, the tablespace cannot contain a compressed table.
8KB	4K (4096)	4
8KB	2K (2048)	2
8KB	1K (1024)	1
4KB	4K (4096)	None. If innodb_page_size is equal to FILE_BLOCK_SIZE, the tablespace cannot contain a compressed table.
4KB	2K (2048)	2
4KB	1K (1024)	1

下面的例子 创建 general 表空间 ， 增加 压缩表。 假设 默认的innodb_page_size 是 16kb。 FILE_BLOCK_SIZE 是8192， 要求压缩表的 KEY_BLOCK_SIZE 是8.

```
mysql> CREATE TABLESPACE `ts2` ADD DATAFILE 'ts2.ibd' FILE_BLOCK_SIZE = 8192
Engine=InnoDB;

mysql> CREATE TABLE t4 (c1 INT PRIMARY KEY) TABLESPACE ts2
ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8;
```

如果在创建 表空间时 不指定 FILE_BLOCK_SIZE，则FILE_BLOCK_SIZE 默认为 innodb_page_size。当FILE_BLOCK_SIZE 等于 innodb_page_size 时，表空间 可能 只包含

未压缩 的 row format (COMPACT、REDUNDANT 和 DYNAMIC) 的 表

Moving Tables Between Tablespaces Using ALTER TABLE

带有 tablespace 选项的 alter table 可以用于 将 表移动到 现有的 general 表空间， 新的 file-per-table 表空间 或 system 表空间。

要从 file-per-table 或 system 表空间 移动 到 general 表空间， 请指定 general 表空间的名称。 general 表空间 必须存在。

```
ALTER TABLE tbl_name TABLESPACE [=] tablespace_name;
```

要从 general 或 file-per-table 移动到 sysmte， 请将 innodb_system 作为 表空间名字。

```
ALTER TABLE tbl_name TABLESPACE [=] innodb_system;
```

要从 system 或 general 移动到 file-per-table。 将 innodb_file_per_table 作为 表空间名字

```
ALTER TABLE tbl_name TABLESPACE [=] innodb_file_per_table;
```

alter table .. tablespace 操作 会 导致 full table rebuild，即使 tablespace 的值 依然是 之前的值。

alter table .. tablespace 不支持 将 表 从 临时表空间 移到 持久表空间。

data directory 子句 允许 与 create table .. tablespace=innodb_file_per_table 一起使用， 但不支持 和 tablespace 选项一起使用。 从8.0.21开始， 在 data directory 子句中指定的 目录 必须 为 innodb 所知。

从 加密的 表空间 移出 表 时 有限制。

Renaming a General Tablespace

```
ALTER TABLESPACE s1 RENAME TO s2;
```

rename general 表空间 需要 create tablespace 的权限。

rename to 始终都是 autocommit 模式。

当 lock tables 或 flush tables with read lock ，lock 住表时， 无法 rename to.

rename tablespace 时， 对 general 表空间内的 表 采用 独占metadata lock， 从而防止并发 ddl。支持并发DML。

Dropping a General Tablespace

drop tablespace

drop 前，必须先删除 表空间中的 所有表。否则 error。
使用下面的 query 来查看 表空间中的表：

```
mysql> SELECT a.NAME AS space_name, b.NAME AS table_name FROM
INFORMATION_SCHEMA.INNODB_TABLESPACES a,
       INFORMATION_SCHEMA.INNODB_TABLES b WHERE a.SPACE=b.SPACE AND a.NAME LIKE
'ts1';
+------------+------------+
| space_name | table_name |
+------------+------------+
| ts1        | test/t1    |
| ts1        | test/t2    |
| ts1        | test/t3    |
+------------+------------+
```

general 表空间不属于 任何特定数据库，drop database 可以删除 属于 通用表空间的 表，但是 不能删除 表空间。

和system 表空间类似， truncate 或 drop general表空间中的表 会在 general 表空间的 .idb 数据文件内部 创建 空闲空间， 这空间只能用于 新的 innodb 数据。 不会像file-per-table 那样 返还给 OS。

MySQL 中 tablespace_name 是 大小写敏感的。

General Tablespace Limitations

1. generated 或 existing 的 表空间 不能 被更改为 general 表空间
2. 不支持 创建 临时 general 表空间
3. general 表空间不支持 临时表。
4. truncate 或 drop 表，只是在 general 表空间的 .ibd 数据文件 内部 创建 空闲空间，这个空闲空间 只能用于 innodb数据。不会还给 OS。 和 system 表空间一样。
   此外，对 shared 表空间 (general 或 system 表空间) 中的 表 执行 alter table 可能增加 表空间 数据文件的 大小。这类操作 需要 和 表+index 一样大的 额外空间。table-copying alter table操作 使用的 额外空间 不会 返还给 OS。
5. general 表空间 不支持 alter table .. discard tablespace 和 alter table .. import tablespace。
6. MySQL5.7.24 开始 不推荐 将 table partitions 放在 general 表空间，从8.0.13 开始删除。
7. 在 源 和 副本 都位于 同一台主机上的 复制环境 中 不支持 add datafile 子句，因为它会导致 源和副本 在同一位置 创建 同名 表空间。 但是如果 省略 add datafile 子句，则会在 数据目录中 创建 表空间，并使用 唯一的 生成文件名，这是可以的。
8. 从MySQL8.0.21 开始， 除非 InnoDB 直接知道， 否则不能在 undo 表空间 目录 (innodb_undo_directory) 中 创建 general 表空间。 已知目录是 由 datadir, innodb_data_home_dir, innodb_directories 定义的。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-undo-tablespaces.html
15.6.3.4 Undo Tablespaces

undo表空间 包含 undo log， 它们是 record 的集合，包含了 如何 undo 事务对 clustered index record 做的 最后的 修改。

Default Undo Tablespaces

mysql实例初始化时 会 创建 2个 默认的 undo 表空间。 默认undo 表空间是在 初始化时 创建的，以便 为 在 接收sql 之前 必须存在的 rollback segments 提供 位置。至少需要 2个 undo 表空间 来支持 undo 表空间的 自动 turncation。

默认的 undo 表空间 是在 innodb_undo_directory 定义的位置创建的。如果没有定义， 则在 data directory 中 创建 默认undo表空间。 默认undo表空间的 数据文件名为 undo_001, undo_002. data directory 中 定义的 相应的 undo 表空间 名字是 innodb_undo_001, innodb_undo_002。

从MySQL 8.0.14 开始， 可以在运行时 使用SQL 创建 额外的 undo 表空间。

Undo Tablespace Size

在MySQL 8.0.23 之前， undo 表空间的 初始大小 取决于 innodb_page_size 值。 对于默认的 16kb page size，初始undo表空间大小是 10mb。对于4kb,8kb,32kb,64kb， 初始undo表空间大小 分别是 7mb,8mb,20mb,40mb。

从MySQL 8.0.23开始， 初始undo 表空间 大小通常是 16mb。

当通过 truncate操作 创建 新的 undo 表空间时，初始大小可能有所不同。这种情况下，如果 file extension size 大于 16mb，且 上次 file 扩展 发生在 1秒内，则 new的 undo 表空间的大小 是 innodb_max_undo_log_size 的 1/4 。

8.0.23之前，undo 表空间一次 扩展 4倍。

8.0.23开始，undo 表空间 至少扩展 16 mb。 如果2次扩展间隔0.1s内，则第二次扩展量 翻倍，最多256mb，如果2次扩展间隔小于0.1s，则第二次扩展量 减少一半，最少16mb。 如果为 undo 表空间 定义了 autoextend_size，则 真正的扩展量是 autoextend_size 和 上面的逻辑 的较大值。

Adding Undo Tablespaces

长事务 会导致 undo 表空间变大， 创建额外的 undo 表空间 可以帮助 防止 单个undo表空间变得太大。

从8.0.14开始，可以在运行时， 使用 create undo tablespace 来创建额外的 undo 表空间。

```
CREATE UNDO TABLESPACE tablespace_name ADD DATAFILE 'file_name.ibu';
```

undo表空间文件名 必须有 .ibu 扩展名。 定义undo 表空间 文件名时 不允许 指定相对 路径。 允许使用 完全限定，但InnoDB 必须知道 该路径。 已知路径 是通过 innodb_directories 定义。 建议使用 唯一的 undo 表空间 文件名，以免在移动 或 克隆 数据时 发生 潜在的 文件名冲突。

。。本节剩余全部。简写了。。

由 innodb_data_home_dir、innodb_undo_directory 和 datadir 变量定义的目录会自动附加

到 innodb_directories 值，无论是否显式定义了 innodb_directories 变量。

If the undo tablespace file name does not include a path, the undo tablespace is created in the directory defined by the innodb_undo_directory variable. If that variable is undefined, the undo tablespace is created in the data directory.

To view undo tablespace names and paths, query INFORMATION_SCHEMA.FILES:

```
SELECT TABLESPACE_NAME, FILE_NAME FROM INFORMATION_SCHEMA.FILES
  WHERE FILE_TYPE LIKE 'UNDO LOG';
```

A MySQL instance supports up to 127 undo tablespaces including the two default undo tablespaces created when the MySQL instance is initialized.

Dropping Undo Tablespaces

```
ALTER UNDO TABLESPACE tablespace_name SET INACTIVE;

DROP UNDO TABLESPACE tablespace_name;
```

The state of an undo tablespace can be monitored by querying the INFORMATION_SCHEMA.INNODB_TABLESPACES table.

```
SELECT NAME, STATE FROM INFORMATION_SCHEMA.INNODB_TABLESPACES
  WHERE NAME LIKE 'tablespace_name';
```

inactive： undo 表空间中的 rollback segment 不再被 新的事务使用。
empty：undo 表空间 是空的，可以被 drop 或 active

Moving Undo Tablespaces

Configuring the Number of Rollback Segments

innodb_rollback_segments 变量 定义了 分配给 每个 undo 表空间 和 全局临时表空间的 rollback segment 数了。 这个可以在 启动 或 运行时 配置
innodb_rollback_segments 默认 128， 这也是最大值。

Truncating Undo Tablespaces
Automated Truncation
Manual Truncation

Expediting Automated Truncation of Undo Tablespaces
To increase the frequency, decrease the innodb_purge_rseg_truncate_frequency setting. For example, to have the purge thread look for undo tablespaces once every 32 times that purge is invoked, set innodb_purge_rseg_truncate_frequency to 32.

```
mysql> SET GLOBAL innodb_purge_rseg_truncate_frequency=32;
```

Performance Impact of Truncating Undo Tablespace Files

Number of undo tablespaces
Number of undo logs
Undo tablespace size
Speed of the I/O subsystem
Existing long running transactions
System load

Monitoring Undo Tablespace Truncation

```
SELECT NAME, SUBSYSTEM, COMMENT FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE
NAME LIKE '%truncate%';
```

Undo Tablespace Truncation Limit

Undo Tablespace Truncation Recovery

Undo Tablespace Status Variables

```
mysql> SHOW STATUS LIKE 'Innodb_undo_tablespaces%';
+----------------------------------+-------+
| Variable_name                    | Value |
+----------------------------------+-------+
| Innodb_undo_tablespaces_total    | 2     |
| Innodb_undo_tablespaces_implicit | 2     |
| Innodb_undo_tablespaces_explicit | 0     |
| Innodb_undo_tablespaces_active   | 2     |
+----------------------------------+-------+
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-temporary-tablespace.html

15.6.3.5 Temporary Tablespaces

innodb 使用 session 临时表空间 和 global 临时表空间

。。简写

Session Temporary Tablespaces

session临时表空间 是在 第一个 创建 磁盘临时表 的 request 时 从 临时表空间pool 中 分配给 session。 一个 session 最多 2个 表空间， 一个用于 用户创建的 临时表，一个用于优化器 创建的 内部临时表。
服务器启动时 会创建一个 包含 10个临时表空间 的 pool。 pool 的大小 永远 不会 缩小。
session 临时表空间 文件 在创建时 大小为 5 pages，并具有 .ibt 文件扩展名。

为session 临时表空间 保留了 40万个 space id。

innodb_temp_tablespaces_dir 变量 定义了 session 临时表空间 创建的位置。 默认是 data directory 的 #innodb_temp 目录 。 如果 临时表空间pool 创建失败，则 启动失败。

```
$> cd BASEDIR/data/#innodb_temp
$> ls
temp_10.ibt  temp_2.ibt  temp_4.ibt  temp_6.ibt  temp_8.ibt
temp_1.ibt   temp_3.ibt  temp_5.ibt  temp_7.ibt  temp_9.ibt
```

The INNODB_SESSION_TEMP_TABLESPACES table provides metadata about session temporary tablespaces.

The INFORMATION_SCHEMA.INNODB_TEMP_TABLE_INFO table provides metadata about user-created temporary tables that are active in an InnoDB instance.

Global Temporary Tablespace
global 临时表空间(ibtmp1) 保存了 对用户创建的 零食表空间 的 修改 的 rollback segment。

innodb_temp_data_file_path 定义了 全局临时表空间 数据文件的 相对路径，名称，大小，属性。 如果没有定义， 默认是 在 innodb_data_home_dir 目录中 创建 一个 名为 ibtmp1 的 自动扩展文件。初始文件大小 略大于 12mb。 重启时 删除并新建 一个 文件。

全局临时表空间不能 驻留在 raw device 上。

INFORMATION_SCHEMA.FILES 提供有关全局临时表空间的元数据。

```
mysql> SELECT * FROM INFORMATION_SCHEMA.FILES WHERE
TABLESPACE_NAME='innodb_temporary'\G
```

To determine if a global temporary tablespace data file is autoextending, check the innodb_temp_data_file_path setting:

```
mysql> SELECT @@innodb_temp_data_file_path;
+------------------------------+
| @@innodb_temp_data_file_path |
+------------------------------+
| ibtmp1:12M:autoextend        |
+------------------------------+
```

检查 全局临时表空间 data file 的大小。

```
mysql> SELECT FILE_NAME, TABLESPACE_NAME, ENGINE, INITIAL_SIZE,
TOTAL_EXTENTS*EXTENT_SIZE
```

```
            AS TotalSizeBytes, DATA_FREE, MAXIMUM_SIZE FROM INFORMATION_SCHEMA.FILES
            WHERE TABLESPACE_NAME = 'innodb_temporary'\G
*************************** 1. row ***************************
      FILE_NAME: ./ibtmp1
TABLESPACE_NAME: innodb_temporary
         ENGINE: InnoDB
   INITIAL_SIZE: 12582912
 TotalSizeBytes: 12582912
      DATA_FREE: 6291456
   MAXIMUM_SIZE: NULL
```

要限制 文件大小：

```
[mysqld]
innodb_temp_data_file_path=ibtmp1:12M:autoextend:max:500M
```

配置innodb_temp_data_file_path 要重启server。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-moving-data-files-offline.html

15.6.3.6 Moving Tablespace Files While the Server is Offline

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-disabling-tablespace-path-validation.html

15.6.3.7 Disabling Tablespace Path Validation

启动时， innodb 会扫描 innodb_directories 定义的 目录 以 查找 表空间文件。 将发现的表空间文件的 路径 根据 data dictionary 中 记录的 path 进行验证，如果不匹配，就更新 data dictionary 中的 path

MySQL8.0.21 引入了 innodb_validate_tablespace_paths 变量 允许禁用 表空间路径 验证。
此功能 适用于 不移动 表空间文件的 环境。
禁用路径验证 可以 缩短 具有 大量表空间文件的 系统 的 启动时间。
如果 log_error_verbosity 是3， 当禁用路径验证时， 启动时 会打印：

```
[InnoDB] Skipping InnoDB tablespace path validation.
Manually moved tablespace files will not be detected!
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-optimize-tablespace-page-allocation.html

15.6.3.8 Optimizing Tablespace Space Allocation on Linux

从MySQL 8.0.22 开始，你可以 优化 linux上 InnoDB 如何 为 file-per-table 和 general

表空间 分配空间。
默认，当需要 额外空间时，InnoDB 将 page 分配给 表空间 并将 null 物理写入到 这些 page。
从MySQL 8.0.22开始，你可以在 linux 上 禁用 innodb_extend_and_initialize 以避免 物理地 将 null 写入到 新分配的 表空间 page 中。 当禁用 innodb_extend_and_initialize 时， 使用 posix_fallocate() 来将 空间 分配给 表空间文件， 该方法 保留 空间 而无需 物理写入 null。

使用 posix_fallocate() 分配page 时，默认情况下 扩展的size 是 small的， 并且 一次通常 只分配 几个page，这可能导致 碎片化 并增加 random IO。
要避免此问题，在启用 posix_fallocate() 时 增加 表空间扩展size。 使用 autoextend_size 选项 来修改 表空间扩展size，这个 最多 4GB。

innodb 在分配 新表空间 page 前 写 redo log 记录。如果 page分配操作 被中断，在 recovery 期间 从 redo log 中 重放该操作。(从redo log 重放的 page 分配操作 会 将 null 物理写入新分配的page)。 分配新page前 写入redo log，而不管 innodb_extend_and_initialize 的配置。

on non-linux systems and windows， innodb 分配新的 page 给 表空间 并将 null 物理写入到 这些page， 是默认行为。 在这些 OS 上 禁用 innodb_extend_and_initialize 会返回以下错误：

> Changing innodb_extend_and_initialize not supported on this platform. Falling back to the default.

。。不过看这个输出，应该是一种警告，不会 中断 启动。 fall back 了。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-tablespace-autoextend-size.html

15.6.3.9 Tablespace AUTOEXTEND_SIZE Configuration
默认下，当 file-per-table 或 general 表空间 需要 额外的 space， 表空间根据下面的 rule 进行 扩展：

1. 如果 表空间 size 小于 一个 extent， 则 一次扩展 一个 page
2. 如果 表空间 size 大于 一个 extent 但是 小于 32个 extent， 一次扩展 一个 extent。
3. 如果 大于 32个 extent， 一次 扩展 4个 extent

从8.0.23开始，通过 指定 AUTOEXTEND_SIZE 可以 指定 file-per-table 或 general 表空间的 扩展量。 配置一个 更大的值，可以帮助 避免 碎片 并促进 大量数据的 摄取 (facilitate ingestion of large amounts of data)

要为 file-per-table 表空间 配置 extension size，在 create table, alter table 语句中指定 autoExtend_size 大小

```
CREATE TABLE t1 (c1 INT) AUTOEXTEND_SIZE = 4M;
```

```
    ALTER TABLE t1 AUTOEXTEND_SIZE = 8M;
```

alter tablespace 不能用于 修改 autoExtend_size

要为 general 配置 extension size， 在 create tablespace, alter tablespace:

```
    CREATE TABLESPACE ts1 AUTOEXTEND_SIZE = 4M;

    ALTER TABLESPACE ts1 AUTOEXTEND_SIZE = 8M;
```

autoExtend_size 也可以用于 undo tablespace, 但是 行为不同。 查看 15.6.3.4

配置的值 必须是 4M 的配置。 否则返回 error

默认为0， 会执行 上面提到的 扩展rule。 （。。page，extent，4extent）

8.0.23中 最大是 64M， 从8.0.24开始最大 4GB

最小的 autoExtent_size 取决于 innodb page size：

InnoDB Page Size	Minimum AUTOEXTEND_SIZE
4K	4M
8K	4M
16K	4M
32K	8M
64K	16M

The default InnoDB page size is 16K (16384 bytes).

```
    mysql> SELECT @@GLOBAL.innodb_page_size;
    +---------------------------+
    | @@GLOBAL.innodb_page_size |
    +---------------------------+
    |                     16384 |
    +---------------------------+
```

对于 file-per-table 表空间 的 table， show create table 只有在 autoExtent_size 非0的时候 才显示它。

```
    mysql> SELECT NAME, AUTOEXTEND_SIZE FROM INFORMATION_SCHEMA.INNODB_TABLESPACES
           WHERE NAME LIKE 'test/t1';
    +---------+----------------+
    | NAME    | AUTOEXTEND_SIZE |
    +---------+----------------+
    | test/t1 |        4194304 |
    +---------+----------------+

    mysql> SELECT NAME, AUTOEXTEND_SIZE FROM INFORMATION_SCHEMA.INNODB_TABLESPACES
           WHERE NAME LIKE 'ts1';
    +------+----------------+
    | NAME | AUTOEXTEND_SIZE |
    +------+----------------+
```

```
| ts1  |         4194304 |
+------+-----------------+
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-doublewrite-buffer.html

15.6.4 Doublewrite Buffer

doublewrite buffer 是一个 存储区域，innodb 在将页面写入 innodb数据文件中的 正确 位置之前 将从 buffer pool flush出来的 page 写入其中。
如果page 写入期间，出现 OS，storage subsystem，意外的mysqld 进程退出， innodb 可以在crash recovery 期间 从 doublewrite buffer 中找到 该page 的副本。

尽管数据被写入了 2次， 但是 双写 缓冲区 不需要 2倍的 IO开销 或 2倍的IO操作。 数据以一个 大的 有序的 块 写入到 双写huffer，with 一次 fsync() OS调用 (除了 innodb_flush_method 设置为 O_DIRECT_NO_FSYNC 的情况)

8.0.20之前，双写buffer 存储区域 是 在 innodb system表空间中。 8.0.20开始，是在 doublewrite files 中。

下面的 变量 提供了 双写buffer 的配置：

1. innodb_doublewrite
   控制是否启用 双写buffer。大多数情况下，默认启用。 要禁用的话，设置为OFF。 如果你更关心 性能 而不是 数据完整性，考虑禁用 双写缓冲区， 例如执行 基准测试时。
   从8.0.30开始，innodb_doublewrite 支持 DETECT_AND_RECOVER 和 DETECT_ONLY 。
   detect_and_recover 和 on 相同。使用这个配置，将完全 启用 双写 buffer，将数据库 page 内容 写入到 双写buffer，在 恢复期间 访问 该buffer 来修复 不完整的 page write。
   使用 detect_only ， 只有 元数据 被写入到 双写buffer。 数据库 page 内容 不写入双写buffer， 恢复时，不使用 双写buffer 来修复 不完整的 page write。 这个 轻量级设置 仅用于 检测 不完整的 页面写入。
   从8.0.30开始， 可以 运行时 修改 innodb_doublewrite 的配置， 在 on,detect_and_recover,detect_and_only 之间切换。 不支持 启用 和 关闭 之间 切换。
   如果 双写buffer 位于 支持原子写的 Fusion-io device，会自动 禁用 双写buffer，并使用 Fusion-io 原子写入 执行 数据文件写入。 但是，请注意 innodb_doublewrite 设置 是全局的。 当双写buffer 被禁用时， 所有的 数据文件 都会被 禁用 双写buffer，包括那些不放在 Fusion-io 硬件上的 文件。 这个功能 只在 Fusion-io 硬件上 可用，并且 仅在 linux 上为 Fusion-io NVMFS 启用。 要充分利用 此功能，建议 innodb_flush_method 设置为 O_DIRECT。

2. innodb_doublewrite_dir

在8.0.20中引入，定义了innodb 创建 双写文件的 目录。如果没有指定 目录，则在 innodb_data_home_dir 目录中 创建 双写文件， 如果 没有指定 则默认为 data directory.
hash symbol '#' 会自动作为 指定目录名称 的前缀， 以 避免和 schema 名字 冲突。 但是，如果 目录名字的 前缀是 . # / ，那么不会 在增加 # 。
理想情况下，双写目录 应该放置在 可用的 最快的 存储介质上。

3. innodb_doublewrite_files
   定义了 双写文件的 数量。 默认下，为 每个 buffer pool instance 创建 2个 双写文件： 一个 flush list 双写文件 和 一个 LRU list 双写文件。

   flush list 双写文件 用于 从 buffer pool flush list 中 flush 的 page。 一个 flush list 双写文件 默认大小是 (innodb page size) * (双写page bytes) 。

   lru list双写 用于 从 buffer pool LRU list 中 flush 的 page。还包含 用于 单page flush 的 slot。 LRU list 双写文件的 默认size 是 (innodb page size) * (双写page + (512 / buffer pool instances数量)), 512 是 single page flushes 保留的slot 总数。

   至少有2个 双写文件， 双写文件 的最大数量 是 buffer pool instance 的数量的 2倍。(buffer pool instance 的数量 通过 innodb_buffer_pool_instance 控制)

   双写文件 的名字 具有以下 格式： (或 如果是 ，那么后缀是)。 例如，下面的 双写文件 是 为 innodb page size 为16kb 且 单个buffer pool 的 mysql 实例 创建的：
   #ib_16384_0.dblwr
   #ib_16384_1.dblwr

   innodb_doublewrite_files 用于 高级性能调整， 默认配置 适合大多数 用户。

4. innodb_doublewrite_pages
   8.0.20中引入， 控制 每个 线程的 最大 双写page 数量。 如果没有配置，就 默认设置为 innodb_write_io_threads。 高级性能调整，默认配置 适用于 大多数用户。

5. innodb_doublewrite_batch_size
   8.0.20 引入，控制 一次batch 中 写入的 双写 page 数量。 高级性能调整，默认配置适用于大多数用户。

从8.0.23开始，innodb 自动加密 属于 加密表空间 的 双写文件page。 同样，属于 page压缩表空间的 双写文件 page 也被压缩。 因此， 双写文件 可以包含 不同的 page type， 包括未加密 和 未压缩的 page， 加密的page，压缩的page， 加密&压缩的page。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-redo-log.html

15.6.5 Redo Log

redo log 是一种 基于磁盘的 数据结构， 用于在 crash recovery 期间 纠正 由不完整 事务 写入的 数据。
正常操作期间，redo log 对 由sql语句或低级api调用产生的修改表数据的 请求 进行编码。 在意外关闭之前 未完成 更新数据文件的 修改 会在 初始化期间，接受connection 之前 自动重做。

redo log 表现为 磁盘上的 redo log 文件。 写入到 redo log文件 的数据 根据 受影响的 record 进行编码，这些数据 统称为 redo。 数据通过 redo log 文件的 passage 表现为 一个不断增加的 LSN 值。 当 数据修改 发生时， redo log 数据被append， 当checkpoint 处理后，老的数据 被truncate。

Configuring Redo Log Capacity (MySQL 8.0.30 or Higher)
从MySQL 8.0.30 开始，innodb_redo_log_capacity 系统变量 控制 redo log文件 占用的 磁盘空间量。你可以在 启动时通过option文件 或 运行时通过set global 来设置。 下面的语句 将 redo log 文件容量设置为 8GB

```
SET GLOBAL innodb_redo_log_capacity = 8589934592;
```

在运行时 设置时， 修改会立即发生，但是可能需要一些时间 才能完全实现 新的limit。如果 redo log 文件 占用的空间 小于 指定值， 脏页 不会 积极地 从 buffer pool flush 到 表空间数据文件， 最终会增加 redo log 占用的磁盘空间。

innodb_redo_log_capacity 取代了 已经弃用的 innodb_log_files_in_group, innodb_log_file_size。 当定义 第一个时， 后2个 被忽略。如果 第一个没有定义，则会用后2个 计算第一个 (group * size = capacity)。 如果都没有设置， redo log 文件容量是默认值即 1004857600 bytes (100MB)， 最大 容量是 128GB

redo log文件保存在 data ditectory 的 #innodb_redo 目录中，除非 innodb_log_group_home_dir 设置了 不同的目录， 如果设置了， 那么 会保存在 那个目录的 #innodb_redo 目录下。
有2种类型的 redo log 文件， ordinary 和 spare (普通和备用)。 普通redo log 文件 是那些 正在被使用的 文件。 备用redo log文件 是那些 等待使用的 文件。
innodb 尝试 总共维护 32个 redo log file，每个 文件的 大小 等于 1/32 * innodb_redo_log_capacity； 但是修改 innodb_redo_log_capacity后， 文件的 大小可能会有所不同。

redo log 文件 命名规范是 #ib_redoN。 N 是 redo log file 编号。备用redo log 文件 由 _tmp 后缀表示。 下面展示了 #innodb_redo 目录中的 redo log 文件，21个活动的， 11个备用的，按顺序编号。

```
'#ib_redo582'  '#ib_redo590'  '#ib_redo598'      '#ib_redo606_tmp'
'#ib_redo583'  '#ib_redo591'  '#ib_redo599'      '#ib_redo607_tmp'
'#ib_redo584'  '#ib_redo592'  '#ib_redo600'      '#ib_redo608_tmp'
'#ib_redo585'  '#ib_redo593'  '#ib_redo601'      '#ib_redo609_tmp'
'#ib_redo586'  '#ib_redo594'  '#ib_redo602'      '#ib_redo610_tmp'
'#ib_redo587'  '#ib_redo595'  '#ib_redo603_tmp'  '#ib_redo611_tmp'
'#ib_redo588'  '#ib_redo596'  '#ib_redo604_tmp'  '#ib_redo612_tmp'
```

'#ib_redo589' '#ib_redo597' '#ib_redo605_tmp' '#ib_redo613_tmp'

每个 普通redo log file 都和 特定范围的 LSN 值 关联。
例如，下面的查询 显示了 上一个示例中 列出的 active redo log file 的 start_LSN,end_LSN 值。

```
mysql> SELECT FILE_NAME, START_LSN, END_LSN FROM
performance_schema.innodb_redo_log_files;
+---------------------------+-------------+-------------+
| FILE_NAME                 | START_LSN   | END_LSN     |
+---------------------------+-------------+-------------+
| ./#innodb_redo/#ib_redo582 | 117654982144 | 117658256896 |
| ./#innodb_redo/#ib_redo583 | 117658256896 | 117661531648 |
| ./#innodb_redo/#ib_redo584 | 117661531648 | 117664806400 |
| ./#innodb_redo/#ib_redo585 | 117664806400 | 117668081152 |
| ./#innodb_redo/#ib_redo586 | 117668081152 | 117671355904 |
| ./#innodb_redo/#ib_redo587 | 117671355904 | 117674630656 |
| ./#innodb_redo/#ib_redo588 | 117674630656 | 117677905408 |
| ./#innodb_redo/#ib_redo589 | 117677905408 | 117681180160 |
| ./#innodb_redo/#ib_redo590 | 117681180160 | 117684454912 |
| ./#innodb_redo/#ib_redo591 | 117684454912 | 117687729664 |
| ./#innodb_redo/#ib_redo592 | 117687729664 | 117691004416 |
| ./#innodb_redo/#ib_redo593 | 117691004416 | 117694279168 |
| ./#innodb_redo/#ib_redo594 | 117694279168 | 117697553920 |
| ./#innodb_redo/#ib_redo595 | 117697553920 | 117700828672 |
| ./#innodb_redo/#ib_redo596 | 117700828672 | 117704103424 |
| ./#innodb_redo/#ib_redo597 | 117704103424 | 117707378176 |
| ./#innodb_redo/#ib_redo598 | 117707378176 | 117710652928 |
| ./#innodb_redo/#ib_redo599 | 117710652928 | 117713927680 |
| ./#innodb_redo/#ib_redo600 | 117713927680 | 117717202432 |
| ./#innodb_redo/#ib_redo601 | 117717202432 | 117720477184 |
| ./#innodb_redo/#ib_redo602 | 117720477184 | 117723751936 |
+---------------------------+-------------+-------------+
```

当进行 checkpoint 时，innodb 保存 checkpoint LSN 到 包含包含这个LSN的 文件的 header 中。在 recovery时， 检查 所有的 redo log file， 并从 最新的 checkpoint LSN 开始 恢复。

提供了几个 状态变量 用于 监控 redo log 和 redo log 的 容量resize操作。

```
mysql> SHOW STATUS LIKE 'Innodb_redo_log_resize_status';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| Innodb_redo_log_resize_status | OK    |
+--------------------------------+-------+

mysql> SHOW STATUS LIKE 'Innodb_redo_log_capacity_resized';
 +---------------------------------+-----------+
| Variable_name                    | Value     |
```

```
+---------------------------------+-----------+
| Innodb_redo_log_capacity_resized | 104857600 |
+---------------------------------+-----------+
```

其他的状态变量

```
Innodb_redo_log_checkpoint_lsn
Innodb_redo_log_current_lsn
Innodb_redo_log_flushed_to_disk_lsn
Innodb_redo_log_logical_size
Innodb_redo_log_physical_size
Innodb_redo_log_read_only
Innodb_redo_log_uuid
```

你可以通过 query innodb_redo_log_files Performance Schema table 来查询 active redo log file 的 信息。

```
SELECT FILE_ID, START_LSN, END_LSN, SIZE_IN_BYTES, IS_FULL, CONSUMER_LEVEL
FROM performance_schema.innodb_redo_log_files;
```

Configuring Redo Log Capacity (Before MySQL 8.0.30)

8.0.30之前，默认，innodb 创建 2个 redo log file 在 data directory, 称为 ib_logfile0, ib_logfile1， 然后 循环写入 这2个文件。

修改 redo log capacity 需要 修改 redo log file 的 size 或 数量， 或both。

1. stop server，确保 正确关闭 (没有error)
2. 编辑 my.cnf ，要更改 redo log file size，则修改 innodb_log_file_size。 要更改文件数量，修改 innodb_log_files_in_group.
3. 启动 mysql server

innodb 检测到 innodb_log_file_size 和 文件size 不同， 就会 写入一个 log checkpoint。关闭，删除 旧 log 文件， 创建 新的 符合 size 的 文件，并使用它。

Automatic Redo Log Capacity Configuration

启用 innodb_dedicated_server 后， innodb 自动 配置 某些 innodb 参数，包括 redo log capacity。 自动配置的配置 适用于 mysql 专用服务器，(mysql 可以使用 所有的系统资源)。

Redo Log Archiving

备份功能 (复制redo log记录) 有时 无法跟上 redo log 的生成速度， 从而导致 由于这些记录被覆盖 而丢失 redo log record。

当 backup 期间 有大量MySQL 服务器 活动，且 redo log file 的存储介质 比 backup 存储介质 更快 时， 通常会出现此问题。

8.0.17 中 引入 redo log archiving feature，通过 将 redo log 记录 顺序 写入 archive 文件 和 redo log 文件 来解决这个问题。 backup 功能 可以根据需要 从 archieve 文件中复制 redo log record， 从而 避免 潜在的 数据丢失。

如果在 服务器上 配置了 redo log archiving。 MySQL 企业版 可以 在 back up mysql

server 时 使用 redo log archiving 功能。

启用 redo log archiving 需要为 innodb_redo_log_archive_dirs 系统设置一个 值。这个值是 分号分隔 的 目录列表， 每个目录格式：label:directory ：

```
mysql> SET GLOBAL
innodb_redo_log_archive_dirs='label1:directory_path1[;label2:directory_path2;…
]';
```

label 是 archive directory 的 任意标识符。 可以是 任意字符串，但不允许使用 冒号 。 可以为空， 但是 依然需要 冒号 分隔。 必须指定 directory_path。 路径可以包含 冒号，但不能有 分号。 激活redo log archiving 时， 路径必须存在。

在激活 redo log archiving 之前，必须配置 innodb_redo_log_archive_dirs 变量。 默认为 null，null表示 不允许 redo log archiving。

。。目录 还有 一些限制。 要独立全新 路径(不能和 其他配置中路径 有 交错)，要权限控制(不能被 全部用户都能访问)。

当支持 redo log archiving 的 实例 开始 backup 时， backup程序 通过 调用 innodb_redo_log_archive_start() 函数 来 激活 redo log archiving

如果你没有 使用 支持redo log archiving 的 backup utility， 也可以手动激活 redo log archiving:

```
mysql> SELECT innodb_redo_log_archive_start('label', 'subdir');
+----------------------------------------+
| innodb_redo_log_archive_start('label') |
+----------------------------------------+
| 0                                      |
+----------------------------------------+
```

or

```
mysql> DO innodb_redo_log_archive_start('label', 'subdir');
Query OK, 0 rows affected (0.09 sec)
```

label 是 innodb_redo_log_archive_dirs 定义中的 label， subdir 是 可选参数，用于 指定用来保存 存档文件的 label标识的 目录的 子目录; 它必须是 简单的目录名称 (不允许 / \ :) 。 subdir 可以 为空，null 或 省略。

只有 具有 INNODB_REDO_LOG_ARCHIVE 权限的 用户 才能通过 调用 innodb_redo_log_archive_start() 激活 redo log archiving, 或使用 innodb_redo_log_archive_stop() 将其 停用。

The redo log archive file path is directory_identified_by_label/[subdir/]archive.serverUUID.000001.log

Note:	激活 redo log archiving (using innodb_redo_log_archive_start()) 的 mysql session 必须在 archiving期间 保持打开状态。 同一session 必须 关闭 redo log archiving (通过 innodb_redo_log_archive_stop())。 如果 session在 redo log archiving显式关闭 之前终止，那么 服务器 会 隐式 停用 redo log archiving 并删除 redo log archive file

在 backup 完成 复制 innodb data file 后，它会停止 redo log archiving 通过 调用 innodb_redo_log_archive_stop() 函数。

如果你没有使用 支持redo log archiving 的 backup ，可以手动 停止：

```
mysql> SELECT innodb_redo_log_archive_stop();
+--------------------------------+
| innodb_redo_log_archive_stop() |
+--------------------------------+
| 0                              |
+--------------------------------+
```

or

```
mysql> DO innodb_redo_log_archive_stop();
Query OK, 0 rows affected (0.01 sec)
```

在 成功停止后， backup程序 从 archive 文件中 查找 redo log 并 复制到 backup 中。

在 backup 程序 完成 复制 redo log ，并不在需要 redo log file archive 后， 它会删除 archive file

Performance Considerations

由于 额外的 写入活动， 激活 redo log archiving 通常造成 小的(minor) 性能消耗。

unix 或 类unix 的 OS 上，如果没有 持续高频update，那么性能影响 通常很小。 window上，性能影响通常更高一点。

如果 持续高频更新， 并且 redo log archiving 和 redo log 文件 在同一个 存储介质上，则 由于复合写入，所以 对性能的影响 可能 更显著。

如果 持续高频更新， 并且 archiving 文件 所在 存储介质 比 redo log file 慢， 则性能受任意影响 (performace is impacted arbitrarily) 。。。。可能很大，可能没有，但只看最坏情况。

redo log archive file 的 write 不会 妨碍 正常的 事务logging， 除非 archive file 的存储介质 比 redo log file 的存储介质 慢很多， 并且 有大量的 持久redo log 积压着 等待写入 redo log 归档文件。 这种情况下， 事务日志 记录 速度 降低到 可以由 redo log archive file 所在的 较慢 存储介质 管理的 级别。

Disabling Redo Logging

从MySQL 8.0.21 开始，你可以使用 alter instance disable innodb redo_log 来 禁用 redo logging

这个功能 旨在 将 数据加载到 新的 MySQL 实例中。 禁用 redo log 可以加速数据加载，因为 避免了 redo log write 和 doublewrite buffering。

警告： 这个功能只用于 加载数据 到 新的 MySQL 实例。 不要在 生产 上 禁用 redo

logging. 禁用redo logging 时， 可以 关闭 和 启动 服务器，但是 如果 意外停止，则可能导致 数据丢失 和 实例损坏。

> [ERROR] [MY-013598] [InnoDB] Server was killed when Innodb Redo logging was disabled. Data files could be corrupt. You can try to restart the database with innodb_force_recovery=6

这种情况下，初始化一个 新的 MySQL 实例， 再次启动 data loading.

enable 和 disable redo logging 需要 INNODB_REDO_LOG_ENABLE 权限(privilege)。

Innodb_redo_log_enabled 状态变量 允许 监控 redo logging 状态。

redo logging 禁用时， 不允许 clone 操作 和 redo log archiving。

ALTER INSTANCE [ENABLE|DISABLE] INNODB REDO_LOG operation requires an exclusive backup metadata lock

下面的过程 演示了 如何 将数据 加载到 新的 MySQL 实例 时 禁用 redo logging

1. 在新 MySQL 实例上， 将 INNODB_REDO_LOG_ENABLE 权限 授予 负责 禁用redo logging 的 用户账户：

   ```
   mysql> GRANT INNODB_REDO_LOG_ENABLE ON *.* to 'data_load_admin';
   ```

2. 使用 data_load_admin 用户， 禁用redo logging

   ```
   mysql> ALTER INSTANCE DISABLE INNODB REDO_LOG;
   ```

3. 检查 Innodb_redo_log_enabled 状态变量 以确保 禁用了 redo logging

   ```
   mysql> SHOW GLOBAL STATUS LIKE 'Innodb_redo_log_enabled';
   +-----------------------+-------+
   | Variable_name         | Value |
   +-----------------------+-------+
   | Innodb_redo_log_enabled | OFF   |
   +-----------------------+-------+
   ```

4. 允许 data load 操作
5. 使用 data_load_admin 用户， 启用 redo logging

   ```
   mysql> ALTER INSTANCE ENABLE INNODB REDO_LOG;
   ```

6. 检查状态，确保启动了

   ```
   mysql> SHOW GLOBAL STATUS LIKE 'Innodb_redo_log_enabled';
   +-----------------------+-------+
   | Variable_name         | Value |
   +-----------------------+-------+
   | Innodb_redo_log_enabled | ON    |
   +-----------------------+-------+
   ```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-undo-logs.html

15.6.6 Undo Logs

undo log 是 和 单个事务关联的 undo log record 的集合。
undo log record 包含 有关 如何 撤销 事务 对 聚集index 记录的 最新修改的 信息。
如果另一个事务需要 将 原始数据 视为 一致读操作的一部分，则从 undo log 中 检索 未 修改的 数据。
undo log 存在于 undo log segments 中， undo log segment 包含在 rollback segments 中。 rollback segment 存在于 undo 表空间 和 global 临时表空间 中。

驻留在 全局 临时 表空间中的 undo log 用于 修改了用户定义的临时表中数据的 事务。这些 undo log 不是 redo-logged，因为它们不是 crash recovery 所必须的。 它们仅用于 服务器运行时 进行回滚。 这种类型的 undo log 通过 避免 redo logging IO 来 提高性能

数据加密，看其他章节中数据 (15.13 InnoDB Data-at-Rest Encryption)

每个undo 表空间 和 全局 临时表空间 分别 支持 最多 128 个 rollback segment。
innodb_rollback_segments 定义了 回滚段 的数量。

一个 rollback segment 支持的 事务数量 取决于 rollback segment中的 undo slot的 数量 和 每个事务 需要的 undo log 的数量。
rollback segment 中 undo slot 的数量 因 Innodb page size 的不同 而不同。

InnoDB Page Size	Number of Undo Slots in a Rollback Segment (InnoDB Page Size / 16)
4096 (4KB)	256
8192 (8KB)	512
16384 (16KB)	1024
32768 (32KB)	2048
65536 (64KB)	4096

。。16k 一个 undo slot

一个事务 最多 分配 4个 undo log。下面的每个类型 最多一种：
1. 用户定义的表 上的 insert
2. 用户定义的表 上的 update 和 delete
3. 用户定义的 零时表 上的 insert
4. 用户定义的 临时表上的 update 和 delete

根据需要 分配 undo log。

在regular table 上执行操作的 事务 被 分配了 来自undo表空间回滚段 的 undo log。 在临时表上执行操作的 事务 被 分配了 来自临时表空间回滚段的 undo log。

分配给 事务的 undo log 在 它的存活期间 保持 attach 到 事务。 例如，为常规表上的 insert 操作 分配给 事务的 undo log 用于 该事务 执行的 常规表 上 所有的 insert 操作。

基于上述因素，以下公式 可以估计 innodb 能够 支持的 并发 读写 事务数。

Note: 在达到 innodb能支持的 并发 读写 事务数 之前，可能会遇到 并发事务限制错误 (concurrent transaction limit error) 。 当分配给 事务的 回滚段 用完 undo slot时，就会 发生这个问题。 在这种情况下，尝试 重新运行 事务。
当事务 对临时表 进行操作时， innodb 能够支持的 并发读写事务 数 收到 全局临时表空间的 回滚段数的 限制，默认 128.
。。是 undo slot 的限制 还是 undo segment 的限制？ 还是说 regular table 是 slot，临时表是 segment ？。。还真是，下面的公式。。

如果每个事务 执行 insert 或 update 或 delete 操作， innodb 能支持的 并发读写数：
(innodb_page_size / 16) * innodb_rollback_segments * number of undo tablespaces

如果每个事务 执行一个 insert 和 一个 update或delete。
(innodb_page_size / 16 / 2) * innodb_rollback_segments * number of undo tablespaces

如果每个事务 在 临时表上 执行insert，并发读写数：
(innodb_page_size / 16) * innodb_rollback_segments

如果 每个事务在 临时表上 执行 insert 和 一个 update或delete。
(innodb_page_size / 16 / 2) * innodb_rollback_segments

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-transaction-model.html

15.7 InnoDB Locking and Transaction Model

要实现 large-scale,busy,or highly reliable 数据库应用， 要从不同的数据库系统移至 大量代码， 要调整MySQL 性能， 那么 理解 Innodb locking， Innodb transaction model 是非常重要的。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html
15.7.1 InnoDB Locking

本节表述了 Innodb 使用的 lock 类型
Shared and Exclusive Locks
Intention Locks
Record Locks
Gap Locks
Next-Key Locks
Insert Intention Locks
AUTO-INC Locks
Predicate Locks for Spatial Indexes

Shared and Exclusive Locks

InnoDB 实现了 标准 row-level locking， 其中有2种类型的锁， shared(s)锁, exlusive(x)锁。

共享锁 允许 持有该锁的 事务 读取一行
独占锁 允许 持有该锁的 事务 更新或删除 行

如果事务T1 在 行 r 上 持有 共享锁，那么 来自另一个事务 T2的 对 行 r 上的 锁的 请求 按如下方式处理

T2 对 共享锁 的请求 可以立即被授予， T1，T2 都在 行 r 上持有 S锁
T2 对 X锁 的请求 不能立即被授予。

如果事务T1 在 行 r 上有 x锁，则无法立即授予 来自不同事务 T2 的 对r 上 任一类型的 锁的请求。 T2 必须等待 事务T1 释放锁

Intention Locks

InnoDB 支持 multiple granularity locking(多粒度锁)，允许 row lock ，table lock 共存。
例如， lock tables .. write 之类的语句 在 指定表上 获得 独占锁。
为了使 多粒度级别的锁 变得实用， innodb 实用 意图锁(intention lock). 意图锁 是 表级锁，它指示 事务 稍后 对 表中的 row 需要 那种 类型的锁 (共享 or 独占)。
有2种类型的 意图锁

intention shared lock (IS) 表示 事务 打算在 表中的 individual(个别) row 上设置 shared lock
intention exclusive lock (IX) 事务打算在表的 individual row 上 设置 独占锁。

For example, SELECT ... FOR SHARE sets an IS lock, and SELECT ... FOR UPDATE sets an IX lock.

intention locking protocol 如下：

在事务可以 在 表中的 一行 上获得 S锁之前，它必须获得 表上的 IS锁 或更强的锁。
在事务 可以获得 表中的一行 的 X锁之前， 它必须获得 表上的 IX 锁。

下面的表格 总结了 表级锁类型的 兼容性

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

如果锁 和 现有锁 兼容，则将锁授予 请求的事务， 但如果 和现有锁 冲突则不会。
事务等待 直到 释放 冲突的 现有锁。
如果锁请求 和 现有锁冲突 并且由于会导致 死锁 而无法授予， 则会发生错误。

除了 全表请求 (如，lock tables .. write) 外， 意图锁不会阻塞任何东西。意图锁的主要目的 是表明 有人正在 锁定一行，或者 要 锁定表中的 一行。
。。 ？ 不会阻塞？ 事务T1 持有 x锁， T2 也想要x锁，但是获得x锁之前，要获得 IX锁，那么 这把 IX 锁不在 T1 手里？ 不在 T1手中的话，IX锁有什么意义？ 难道 是 为了 让 申请锁 这个动作 串行执行？ 防止 T2，T3 同时 申请 X 锁？ 为了让 申请X锁 变成 原子性的？ 。。。那 申请意图锁 岂不是还要加个 锁 来让 申请IX锁 变得原子性。。 申请锁 本身就是 原子性的吧。 不需要 加个 意图锁 来 原子性。
。。。 上面的表格里 写了 IX 和 IX 是 compatible的。 就是 读锁一样， 大家都可以获得。。 真的只是 "想" 而已。。

意图锁 的事务数据 在 show engine innodb status 和 innodb monitor 输出 中类似 下面的内容：

```
TABLE LOCK table `test`.`t` trx id 10080 lock mode IX
```

Record Locks

record lock 是对 index record 的lock。 例如，SELECT c1 FROM t WHERE c1 = 10 FOR UPDATE; 防止 任何其他事务 insert,update,delete t.c1 为 10的 row。
。。for update 也是一个 IX lock

record lock 总是 lock index record， 即使 table 没有index。 对于这种情况，innodb 创建一个 隐式的 聚集index 并使用这个 index 来 锁定record。

transaction data for record lock 在 show engine innodb status 和 innodb monitor 输出中 类似 下面的内容

```
RECORD LOCKS space id 58 page no 3 n bits 72 index `PRIMARY` of table `test`.
`t`
trx id 10078 lock_mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 8000000a; asc     ;;
 1: len 6; hex 00000000274f; asc     'O;;
 2: len 7; hex b60000019d0110; asc        ;;
```

Gap Locks

gap lock 是 index record 之间的 gap 上的 lock， 或在 第一条record 之前， 或 最后一条 record 之后的 gap 上的lock。
例如 SELECT c1 FROM t WHERE c1 BETWEEN 10 and 20 FOR UPDATE; 防止其他事务 将值 15 插入到 t.c1列。

一个gap 可能 跨越 单个index value， 多个index value，甚至是空的。

gap lock 是 性能和并发 之间 权衡的 一部分， 并且 (只)用于 某些事务隔离级别。

使用 unique index 来 lock row 以搜索 unique row 的 语句 不需要 gap lock。(这不包括搜索条件 仅包括 唯一index 的某些列 的情况，这种情况下， 确实会发生 gap lock)。
例如，如果id 列具有 唯一index， 则下面的 语句仅使用 id=100 的 row 的 index record

lock， 不关心 其他session 是否在 前面的gap 中插入 row。

```
SELECT * FROM child WHERE id = 100;
```

如果 id列 没有 index 或 具有非唯一index， 则该语句会 锁定前面的(preceding) gap。

还需要注意的是，不同的事务可以在 间隙上 持有 冲突的 lock。 例如，事务A可以在 gap 上持有一个 共享间隙锁(gap s-lock)，同时 事务B 在同一个间隙上 持有 一个 排他性间隙锁(gap x-lock)。 允许 冲突间隙锁 的原因是，如果 从 index 中 purge record，则必须 合并不同事务 在 记录上 持有的 间隙锁。

innodb 的 gap lock 是"purely inhibitive(纯粹的抑制性)"，这意味着它们的 唯一目的 是防止 其他事务插入到 间隙中。 间隙锁 可以共存。 一个 事务 采用的 gap lock 不会阻止另一个事务 在 同一个 gap 上 使用 gap lock。 共享和独占 gap lock 之间 并没有区别。它们彼此不冲突，并且 执行相同的 功能。

可以 显示 禁用gap lock。 如果你将 事务隔离级别 设置为 read committed，就会发生这种情况。 这种情况下，gap locking 对 search 和 index scan 禁用， 仅用于 外键约束检查 和 重复键 检查。

使用 read committed 隔离级别 还有 其他影响。 在mysql 评估 where条件后， 不匹配 row 的 record lock 会被释放。 对于 update 语句，innodb 执行 "semi-consistent" read，这样它会将 最新提交的版本 返回给 MySQL， 以便mysql可以确定 该行 是否 匹配 update 的 where条件。

Next-Key Locks

next-key lock 是一个 组合， 是 index record 上的 record lock 和 在这个idnex record 之前的 gap 的 gap lock 的组合。

innodb 执行 row-level locking 的方式是， 当它 搜索 或 扫描 table index时，它会在 它遇到的 index record 上 设置 共享 或 独占 锁。因此 行级锁 实际上是 index record 锁。在 index record 上的 next-key lock 也会影响 index record 前面的 gap。 即， next-key lock 是 index record lock + index record 前面的 gap 的gap lock 。 如果一个session 在 index record R 上有 共享或排他锁，则另一个 session 不能在 索引顺序 中 R 之前的间隙 插入 新的 index record。

假设index 包含 值10,11,13,20。该index 可能的 next-key lock 可能涵盖以下区间， 其中 圆括号 表示 排除 区间断点，[]表示 包括 端点：

```
(negative infinity, 10]
(10, 11]
(11, 13]
(13, 20]
(20, positive infinity)
```

对于最后一个 interval， next-key lock 锁定 index中最大值 上方的 gap。

默认下，innodb 以 repeatable read 事务隔离级别 运行。这种情况下， innodb 使用 next-key lock 进行 搜索 和 index scan， 这可以防止 phantom row (幻象行)

next-key lock 的 transaction data 在 show engine innodb status 和 innodb 监控器 中显示 类似下面的 内容：

```
RECORD LOCKS space id 58 page no 3 n bits 72 index `PRIMARY` of table `test`.
`t`
trx id 10080 lock_mode X
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;

Record lock, heap no 2 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 8000000a; asc     ;;
 1: len 6; hex 00000000274f; asc     '0;;
 2: len 7; hex b60000019d0110; asc        ;;
```

Insert Intention Locks

insert intention lock 是一种 在 插入行之前 由 insert 操作 设置的 gap lock。

这个lock 表示 插入的意图，即 如果 插入到 同一个 index gap 中的 多个 事务 没有插入到gap中的 同一位置，则它们无需 相互等待。

假设有值 为 4 和 7 的 index record。 分别尝试 插入 值5 和6 的单独事务，在获得 插入行的 排它锁之前， 每个事务 通过 insert intention lock lock了 4和7之间的 gap， 但不会 相互阻塞，因为 row 是不冲突的。

下面演示了 在获得 被插入的 record 的 独占锁 之前 获取 插入意图锁 的 事务。 这个例子涉及 2个客户端 A和B。

客户端A 创建一个 包含 2条index record (90,102) 的表，然后启动一个事务， 该事务 排它锁 放在 id大于100的 index record 上，排它锁 包括 102之前的 gap lock。

```
mysql> CREATE TABLE child (id int(11) NOT NULL, PRIMARY KEY(id)) ENGINE=InnoDB;
mysql> INSERT INTO child (id) values (90),(102);

mysql> START TRANSACTION;
mysql> SELECT * FROM child WHERE id > 100 FOR UPDATE;
+-----+
| id  |
+-----+
| 102 |
+-----+
```

客户端B 开始事务 以将 记录插入到 gap 中。 事务在等待 获得 独占锁 期间采用 插入意向锁。

```
mysql> START TRANSACTION;
mysql> INSERT INTO child (id) VALUES (101);
```

插入意图锁的 transaction data 在 show engine innodb status 和 innodb 监视器 输出 中显示 类似于 下面的内容：

```
RECORD LOCKS space id 31 page no 3 n bits 72 index `PRIMARY` of table `test`.
`child`
trx id 8731 lock_mode X locks gap before rec insert intention waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 80000066; asc    f;;
 1: len 6; hex 000000002215; asc     " ;;
```

2: len 7; hex 9000000172011c; asc   r ;;...

AUTO-INC Locks
auto-inc lock 是一种 特殊的 表级锁， 被 那些插入到 具有auto_increment列的表中 的 事务 使用。
最简单的情况下，如果一个事务 正在 向 表中插入值，则任何其他 事务 都必须等待，这样第一个事务的 row insert 收到 连续的 主键值。

innodb_autoinc_lock_mode 变量 控制 用于 auto-increment locking 的算法。 它允许你选择如何 在可预测的 自动增量序列 和 插入操作的 最大并发性 之间 进行权衡。

Predicate Locks for Spatial Indexes
innodb 支持 包含 spatial data 的 列的 spatial index。
为了处理 涉及 spatial index 的操作的 lock ， next-key lock 不能很好地支持 repeatable read 或 serializable 事务隔离级别。 多维数据 没有绝对的 排序概念，所以不清楚 哪个 是 "next" key。

为了支持 具有 spatial index 的 隔离级别，innodb 使用 predicate lock。
spatial index 包含 最小边界矩形(minimun bounding rectangle MBR) 值， 因此 InnoDB 通过在 用于查询的 MBR 值上设置 谓词锁 来强制 对 index 进行一致性读取。 其他事务 无法插入或修改 和查询条件匹配的 row。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-model.html
15.7.2 InnoDB Transaction Model

15.7.2.1 Transaction Isolation Levels
15.7.2.2 autocommit, Commit, and Rollback
15.7.2.3 Consistent Nonlocking Reads
15.7.2.4 Locking Reads

innodb 事务模型 旨在 将 多版本(multi-version)数据库的 最佳属性 和 传统的 两阶段锁定(tow-phase locking) 相结合。
innodb 在 行级别 执行 locking， 并在 默认情况下 以 非锁定一致读取 的形式 运行查询，以oracle style。
innodb 中的 锁信息 以节省空间的 方式存储，因此不需要锁升级。
通常允许多用户 锁定 innodb表中的 每一行，或行的 任何随机子集， 而不会导致 innodb 内存耗尽。
。。。？

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html

15.7.2.1 Transaction Isolation Levels

事务隔离级别是 数据库处理的 基础之一。
隔离级别 是在 多个 事务 同时 进行更改 和 执行查询时 微调 性能 与 结果的可靠性，一致性 和 可再现性 之间的 平衡的设置。

innodb 提供 SQL:1992 标准表述的 所有 4个事务隔离级别：READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。 默认是 REPEATABLE READ。

可以使用 set transaction 语句 修改 单个session 或 所有后续connection的 隔离级别。
要为 所有connection 设置 服务器 级别的默认 隔离级别， 请在 option 文件 或 命令行 中使用 --transaction-isolation 选项。

innodb 使用 不同的 locking strategy 支持 上面描述的 每个 事务隔离级别。
对于acid 合规性 很重要的 关键数据的操作，可以使用 默认的 repeatable read。 你可以使用 read committed 甚至 read uncommitted 来放宽 一致性规则， 在 比如 bulk reporting 的情况下， 精确的 一致性 和 可重复的 结果 不如 最小化 locking 开销 重要。
serializable 比 repeatable read 更严格，主要用于 特殊情况，如 XA事务，及解决并发和死锁问题。

下面的列表 描述了 mysql 如何支持 不同的 事务级别， 该列表 从 最常用的 级别 到 最少用的 级别：

repeatable read

innodb 的默认隔离级别。 同一个事务中的 consistent read(一致性读) 读取第一次读取 时建立的 snapshot。 这意味着，如果你在同一个事务中 发出多个 普通(nonlocking) select，这些select 相互一致。
对于locking read (带有 for update, for share 的 select)， update 和delete 语句，locking 依赖于 语句是使用 具有唯一搜索条件的 唯一index 还是 范围类型的搜索条件。
对于 具有unique search condition 的 unqiue index， innodb 只锁定 找到的 index record， 而不锁定它之前的 gap
对于其他搜索条件，innodb 锁定 扫描的 index range， 使用 gap lock 或 next-key lock 来 block 其他 session insert 到 gap 中。

READ COMMITTED

每次 consistent read(一致性读)， 即使在 同一个事务中，也会设置 并读取自己的 新快照。
对于 locking read，update，delete， innodb仅锁定 index record，而不lock 它们之前的 gap， 因此 允许 在 locked record 旁边自由插入 新记录。 间隙锁 仅用于 外键约束检查 和 重复键检查。
由于 gap locking 已禁用，因此 可能出现 phantom row (幻象行)。
read committed 隔离级别 只支持 基于row 的 binary logging。 如果你使用 read committed 和 binlog_format=MIXED， 服务器自动使用 基于row 的binary logging。

Using READ COMMITTED has additional effects:
对于 update，delete 语句，innodb 仅对其 更新或删除的row 持有锁。 在MySQL 评估

where 条件后， 不匹配的row 的 记录锁 被释放。 这大大降低了 死锁的 可能性，但仍然可能发生

对于update 语句，如果 一行已经被锁定， innodb 执行 "semi-consistent" read (半一致性读取)，将最新提交的 版本 返回给 mysql，以便mysql 可以确定 该行 是否匹配 update 的where 条件。 如果 row 匹配 (必须更新)，mysql 再次读取该行，这一次 innodb 要么 lock 它，要么 等待 lock 它，

下面的例子，从下面的表开始：

```
CREATE TABLE t (a INT NOT NULL, b INT) ENGINE = InnoDB;
INSERT INTO t VALUES (1,2),(2,3),(3,2),(4,3),(5,2);
COMMIT;
```

在这种情况下，表没有index， 因此 搜索和 index scan 使用 隐式的 聚簇index 用来 record locking 而不是 indexed column

。。。？ 之前一直说 锁 index record， 那么这个 index 是聚簇index 还是 二级idnex？ 如果 是二级 index ，我换一个 搜索条件 不就可以跳过了？ 所以 感觉 必然是 锁 聚集 index， 但是这里说 而不是 indexed column。。 。。 锁定的 index record， index 到底怎么来的？ 是根据 where 子句来的？ 然后 会 走到 聚集index？ 但是 二级index 应该直接指向 row 吧？ 不过 聚集index 和 row 实际上差不多 (应该在一起的)。

假设一个session 使用下面的语句 执行 update：

```
# Session A
START TRANSACTION;
UPDATE t SET b = 5 WHERE b = 3;
```

还假设 第二个 session 在 第一个session之后执行这些 语句 来执行 update：

```
# Session B
UPDATE t SET b = 4 WHERE b = 2;
```

当innodb 执行 每个 update时， 它首先为 每个 row 获得 一个 独占锁，然后决定是否修改它。如果不修改，就释放锁。 否则 保留锁 直到 事务结束，这会 影响 事务处理，如下所示：

```
x-lock(1,2); retain x-lock
x-lock(2,3); update(2,3) to (2,5); retain x-lock
x-lock(3,2); retain x-lock
x-lock(4,3); update(4,3) to (4,5); retain x-lock
x-lock(5,2); retain x-lock
```

第二个update 在尝试获得 任何锁 时 立即 阻塞 (因为第一个更新 在 所有的 row 上都 保留了 锁)， 并且 在第一个 update 提交 或 回滚之前 不会继续：

```
x-lock(1,2); block and wait for first UPDATE to commit or roll back
```

如果改为 使用 read committed，则第一个 update 在它读取的 每一行上 获得一个 x锁，并为它 不修改的 row 释放这些锁。

```
x-lock(1,2); unlock(1,2)
x-lock(2,3); update(2,3) to (2,5); retain x-lock
x-lock(3,2); unlock(3,2)
x-lock(4,3); update(4,3) to (4,5); retain x-lock
x-lock(5,2); unlock(5,2)
```

。。？那它上面使用了 什么 事务隔离级别？ repeatable commit？ 应该是的， repeatble read 如果 不是 unique index 上的 unique search ，那么 就有 gap locking.

但是，如果where条件 包含 indexed 列，并且 innodb 使用 index，则在获取 和 保留 record lock 时 只考虑 indexed 列。在下面的例子中，第一个update 在 b=2 的 每一行上 获取 并保留 一个 x 锁。 第二个update 尝试 获取相同记录上的 x锁时 阻塞， 因为它 还使用 在 列b上定义的index。

```
CREATE TABLE t (a INT NOT NULL, b INT, c INT, INDEX (b)) ENGINE = InnoDB;
INSERT INTO t VALUES (1,2,3),(2,2,4);
COMMIT;

# Session A
START TRANSACTION;
UPDATE t SET b = 3 WHERE b = 2 AND c = 3;

# Session B
UPDATE t SET b = 4 WHERE b = 2 AND c = 4;
```

。。index。 那么 我换一个sql 不走这个index ，那么 岂不是 锁不住？ 和之前 红色一样。。

The READ COMMITTED isolation level can be set at startup or changed at runtime. At runtime, it can be set globally for all sessions, or individually per session.

READ UNCOMMITTED

select 语句 以 nonlocking 方式 执行， 但可能 使用 row 的 早期版本。 因此，使用这个 隔离级别，这样的 读取是 不一致的。 这也称为 脏读。 否则，此级别的 工作方式 类似 read committed.

SERIALIZABLE

类似于 repeatable read，但如果 禁用自动提交，innodb 会将所有的 select 语句 隐式 转换为 select .. for share。 如果启用了 自动提交，则 select 是它自己的事务。 因此，它是只读的，并且 如果 作为一致 (非locking) 读取 执行 并且 不会阻塞 其他事务，则可以序列化。(如果其他事务 修改了 选定的 row，要强制一个 普通的 select 阻塞，禁用 autocommit)

This level is like REPEATABLE READ, but InnoDB implicitly converts all plain SELECT statements to SELECT ... FOR SHARE if autocommit is disabled. If autocommit is enabled, the SELECT is its own transaction. It therefore is known to be read only and can be serialized if performed as a consistent (nonlocking) read and need not block for other transactions. (To force a plain SELECT to block if other transactions have modified the selected rows, disable autocommit.)

从 MySQL 8.0.22 开始，从 MySQL 授权表（通过连接列表或子查询）读取数据但不修改它们的 DML 操作不会在 MySQL 授权表上获取读取锁，无论隔离级别如何。

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-autocommit-commit-rollback.html
15.7.2.2 autocommit, Commit, and Rollback

在innodb中，所有用户活动 都是发生在 事务中的。
如果 启用了 autocommit，每个sql 语句都会形成一个事务。
默认情况下，mysql 对connection 的每个session 启用 autocommit。 所以，如果语句没有返回 error 的话，mysql会在 每个sql 执行后 commit。 如果返回错误，则提交 或 回滚 取决于 error。

启用了 自动提交的 session 可以 通过 显式 start transaction 或 begin 语句 开始， 并以 commit 或 rollback 语句 结束 来 执行 多语句事务。

如果 在 session 上 通过 set autocommit=0 来 禁用 autocommit，则 该session 始终打开事务。 commit 或 rollback 会结束 当前事务 并 开始一个新的。
。。。6
。。所以 最开始说 所有用户活动 都是 事务中的， 。。。 没办法 无事务执行。。 无事务实际上 等于 每个sql 一个事务。

如果 禁用了 autocommit， 且没有 显式 commit，那么 mysql 会 rollback。

一些语句 隐式 地 结束一个 事务 (。。应该算是前一个事务，也不是，。)， 就像 你 在 执行语句前 执行 commit 了。 查看 13.3.3
。。https://dev.mysql.com/doc/refman/8.0/en/implicit-commit.html
。。DDL 语句。 还有 load data， analyze table， create index。 start replica 等等。。。日常使用(应该)碰不到。

commit 意味着 在 当前事务中 所做的 修改 是永久的， 并对 其他session 可见 (。。还要看隔离级别吧。后来的session肯定可以。)。 另一方面rollback 取消 当前事务 所做的 所有修改。
commit 和 rollback 都会 释放 在当前事务期间 设置的所有 innodb lock

Grouping DML Operations with Transactions
默认下，与mysql 服务器 的connection 以 autocommit 模式 开始，它会 在你 执行每个sql 时 自动提交它。 其他数据库的 标准做法是 发出一系列 DML 并 将它们 一起 提交 或 回滚。

要使用 多语句 事务， 请使用 sql 语句 set autocommit=0 关闭autocommit， 并根据需要使用 commit 或 rollback 结束每个事务。
to leave autocommit on ， 请以 start transaction 开始， rollback ,commit 结束。
下面的例子展示了 2个 事务， 第一个 commit 了， 第二个 rollback 了。

```
$> mysql test

mysql> CREATE TABLE customer (a INT, b CHAR (20), INDEX (a));
Query OK, 0 rows affected (0.00 sec)
mysql> -- Do a transaction with autocommit turned on.
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO customer VALUES (10, 'Heikki');
Query OK, 1 row affected (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
mysql> -- Do another transaction with autocommit turned off.
mysql> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO customer VALUES (15, 'John');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO customer VALUES (20, 'Paul');
Query OK, 1 row affected (0.00 sec)
mysql> DELETE FROM customer WHERE b = 'Heikki';
Query OK, 1 row affected (0.00 sec)
mysql> -- Now we undo those last 2 inserts and the delete.
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM customer;
+------+--------+
| a    | b      |
+------+--------+
|   10 | Heikki |
+------+--------+
1 row in set (0.00 sec)
mysql>
```

Transactions in Client-Side Languages

。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html

15.7.2.3 Consistent Nonlocking Reads

一致性读取(consistent read)意味着 innodb 使用 multi-versioning 来 给 一个 query 展现 数据库 在 某个时间点的 快照。 query 会看到 时间点之前commit 的 事务的 修改， 时间点之后 commit 的 修改 看不到。 这个规则的例外是 查询 会看到 同一个事务 先前语句所做的 修改。 这个例外可能导致 下面的 问题：如果你更新表中的 某些row，则 select 会看到 updated row 的最新版本(。。??)， 但它也可能会看到 任何row 的 旧版本。 如果 其他 session 同时更新 同一个表，则 意味着 你可能会看到 该表 处于 数据库中 从未存在过的状态。

。。。不太清楚 这个 最新版本 是指 时间点 之后 其他事务 commit的 数据 + 自己本次更新的数据 吗？ 。。 但是 前面说的 例外是 同一个事务 先前语句的 修改。 所以 时间点后 commit 的 是不可能看到的。 还有最后一句 从未存在过的状态。。 都 update 了， 肯定从未存在过啊。。。

。。下面提到了 不会感知到 其他事务 在 时间点后的 操作。

如果事务隔离级别为 repeatable read (默认级别)， 则 同一事务中的 所有 一致性读取 都读取 第一次 read 时 建立的 快照。
。。这个有没有上面的 例外。。应该有吧。。

read committed 级别， 事务中的 每个 一致性读取 都会 设置 并读取 自己的 新快照。

一致性读取 是 innodb 在 read committed 和 repeatable read 隔离级别 处理 select 语句的 默认模式。 一致读取 不会 对其 访问的 表 设置任何 lock，因此 其他session 可以在对表 进行 一致读取的 同时 自由修改这些表。

假设你在默认的 repeatable read 隔离级别下运行。 当你发出 一致读取 (即 普通的select 语句)，innodb 会为你的 事务提供一个 时间点(timepoint) 用于 确定 你的query 看到的 数据库 内容。 如果另一个 事务 在 你的 时间点 之后 删除row 且 commit，你 不会 感知到 删除行为。 insert 和 update 类似。

Note：
数据库状态的 快照 适用于 事务中的 select， 不一定适用于 DML 语句。
如果你 insert 或 modify 一些row ，然后 commit， 则 从另一个并发 的 repeatable read 事务 发出的 delete 或 update 可能会 影响 那些 刚提交的 row， 即使 session 无法查到它们。 如果 一个事务 确实 更新 或 删除了 由 不同事务提交的row ，那么这些 更改 对 当前 事务 是可见的。
例如，你可能遇到下面的情况：

```
SELECT COUNT(c1) FROM t1 WHERE c1 = 'xyz';
-- Returns 0: no rows match.
DELETE FROM t1 WHERE c1 = 'xyz';
-- Deletes several rows recently committed by other transaction.
```

。。查询没有， 但是 其他 事务 在 查询后 commit 了 几条 符合条件的 数据， delete 会把刚提交的 给删除掉。

```
SELECT COUNT(c2) FROM t1 WHERE c2 = 'abc';
-- Returns 0: no rows match.
UPDATE t1 SET c2 = 'cba' WHERE c2 = 'abc';
-- Affects 10 rows: another txn just committed 10 rows with 'abc' values.
SELECT COUNT(c2) FROM t1 WHERE c2 = 'cba';
-- Returns 10: this txn can now see the rows it just updated.
```

。。查询没有， 但是 其他事务 在 查询后 commit 了 10条数据， update 会 修改 这10条数据， 然后 查询 就有了。

。。。就是说 update 和 delete 会 搜索 最新的 数据。不走 快照。
。。好像也是啊， 4个隔离级别 3个都是 read 相关， 和 update delete 完全无关。。

你可以通过提交事务 然后 执行 另一个 select 或 start transaction with consistent snapshot ， 来 刷新 时间点。

This is called multi-versioned concurrency control.
。。。？？？

在下面的例子中， session A 看到 B insert 的row 只有 当 B committed 且 A也committed， 因此 timepoint 被 提前了(advance) by B的commit （ so that the timepoint is advanced past the commit of B)

```
             Session A              Session B

           SET autocommit=0;      SET autocommit=0;
time
|          SELECT * FROM t;
|          empty set
|                                 INSERT INTO t VALUES (1, 2);
|
v          SELECT * FROM t;
           empty set
                                  COMMIT;

           SELECT * FROM t;
           empty set

           COMMIT;

           SELECT * FROM t;
           ---------------------
           |    1    |    2    |
           ---------------------
```

如果你想看 数据库的 "最新" 状态， 使用 read committed 隔离级别 或 locking read
    SELECT * FROM t FOR SHARE;

使用 read committed 隔离级别， 事务中的 每个 一致性读取 都会 设置 并读取 自己的 新快照。 使用 for share ，会发生 locking read: select 阻塞 直到 包含最新 行 的事务 结束。 (15.7.2.4 Locking Reads 。。下一节)

一致性读 不适用于 某些 DDL 语句：

1. drop table
2. alter table

对于未指定 for update 或 for share 的子句， 如 insert into .. select, update .. (select)， create table 。。 select， read 的type 会有所不同：

1. 默认下，innodb对这些语句 使用 更强的锁，并且 select 部分的 行为类似于 read committed，即 每个 一致读取，即使在 同一个事务中， 也会设置 和 读取自己的 新快照。
2. 要在这种情况下 执行 nonlocking read, 请将事务的隔离级别 设置为 read uncommitted 或 read committed， 以避免 对 选定表 读取的row 设置 lock

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html

15.7.2.4 Locking Reads

如果你 query数据 然后在 同一个事务中 插入 或 update 这些数据，则 常规的 select 无法提供 足够的 保证。 其他事务 可以 更新或删除 你刚查询到的 row。

innodb 支持 2种 类型的 locking read， 用于提供额外的安全性：
select .. for share
在读到的 所有row 上 设置 shared mode lock。 其他会话 可以读取这些row，但是 在你 的事务 commit 之前 无法修改它们。 如果这些被读到的row 中 有 row 被 另一个 还没有 commit 的 事务 change 了，那你的query 会等待 那个事务结束，然后 使用 最新值。

select .. for share 是 select .. lock in share mode 替代品，但 lock in share mode 依然 向后兼容， 两者是 等价的。 但是 for share 支持 of table_name， nowait，skip locked 选项

在MySQL 8.0.22 之前， select .. for share 需要 select 权限 和 至少一个 delete,lock tables, update 权限， 从 8.0.22 开始，只需要 select 权限。

8.0.22开始，select .. for share 不会在 mysql 授权表 上 获取 读锁。

select .. for update

对于搜索 遇到的 index record， lock row 和 任何关联的index entries，就像你为这些row 发出 update 语句一样。 其他事务 被 阻止更新 这些 row ，执行 select .. for share 或读取 某些事务隔离级别 的数据。 一致读取 忽略 read view 上的 record 上的 任何lock (旧版本的 record 不能被 lock；它们是通过 在 记录的 内存副本上 应用 undo log 来重建的)

需要select 权限 和 至少一个：delete，lock tables,update 权限。

这些子句 主要在 处理 树结构 或 图形结构数据 时非常有用，无论是在 单个表中 还是在 多个表中 拆分。 你从一个地方 遍历 边缘 或 树枝 到另一个地方，同时 保留 并更改 任何 这些 pointer 值的权利。
。。。。。。

当事务提交或回滚时， 由 for share 和 for update query 设置的 所有lock 都被 释放。

Note：
只有 autocommit 被禁用(通过 start transaction 或 设置autocommit 为0)时， 才能 locking read 。

outer 语句中的 locking read 子句 不会 锁定 子查询中表的row。 除非 subquery 中 还指定了 locking read。
例如，下面的不会lock t2 的row

```
SELECT * FROM t1 WHERE c1 = (SELECT c1 FROM t2) FOR UPDATE;
```

要lock t2 的数据，需要 加 locking read 子句

```
SELECT * FROM t1 WHERE c1 = (SELECT c1 FROM t2 FOR UPDATE) FOR UPDATE;
```

Locking Read Examples

假设你要在 child 表 插入新row，并确保 child row 在 parent表中有个 parent row。 你的应用代码 可以确保 整个操作序列的 引用完整性。

首先，使用 一致性读取 query parent 表， 并验证 parent row 是否 存在。 你可以安全的将 child row insert 到 child 表吗？ 不， 因为 其他session 可能在 你的select 和 insert 之间 删除 parent row， 而你却不知道。

为了避免这个问题，请使用 for share

```
SELECT * FROM parent WHERE NAME = 'Jones' FOR SHARE;
```

for share 查询到 parent 'Jones' 后， 你可以 安全地 将 chile row 插入到 child 表 并commit事务。 任何 尝试 在 parent中 适当row 上 获得 独占锁的 事务 都会 等待，直到你完成， 即，直到 所有表中的 数据 处于 一致状态。

再举一个例子， 考虑 表 child_codes ，有一个 integer counter 列，用于 给 添加到 child 表的 每个 child 分配一个 唯一标识符。 不要 使用 一致性读 或 for share读， 因为 数据库的2个用户 可能会看到 相同的 计数器值，并且 如果 2个 事务 尝试 添加 child 的 标识符时， 标识符会相同， 会发生 duplicate-key error。

在这里，for share 不是一个 好的 解决方案，因为如果 2个用户 同时 读取 counter，则 至少有一个用户 在尝试 更新 counter 时 会 deadlock。

要实现 counter 的读取和 递增， 首先用 for update 来 locking read counter，然后增加值：

```
SELECT counter_field FROM child_codes FOR UPDATE;
UPDATE child_codes SET counter_field = counter_field + 1;
```

select .. for update 读取 最新可用数据，在它读取 的每一行上 设置 独占锁。 即， 它设置和 update 语句一样的 锁 到 row 上。

前面的描述只是 select .. for update 如何工作的 一个例子，在MySQL 中，生成 唯一标识符的 具体任务 实际上可以通过 对表的 一次访问 来完成：

```
UPDATE child_codes SET counter_field = LAST_INSERT_ID(counter_field + 1);
SELECT LAST_INSERT_ID();
```

。。一条语句，原子性。 不， 上面说了 和 update 一样的 lock ， 都是独占锁 所以 一条 update 执行的时候 通过 独占锁 来完成 原子性。

select 语句仅retrieve 标识符 信息（特定于 当前 connection），它不访问 任何表。

Locking Read Concurrency with NOWAIT and SKIP LOCKED
如果 row 被 事务 lock， 则 request 被lock 的row 的 select .. for update/share 事务 必须等待 直到 那个事务释放lock。
这个行为可以 防止 事务更新 或 删除 那些 其他事务query for update的 row。 但是，如果你希望 query 在 请求的row 被lock 时 立即返回，或者 从resultset 中排除 locked row 是可以接受的， 则无需等待 lock 释放。

为了避免 等待 其他 事务 释放 row lock， nowait 和 skip locked 可以 和 select .. for update/select locking read 语句 一起使用。

nowait
使用 nowait 的 locking read 从不等待 获取 row lock。 query 立即执行， 如果 请求的 row 被lock ，则 抛出 error。

skip locked
使用 skip locked 的 locking read 不会 等待 row lock， 立即执行， 从result set 中 排除 被lock 的row。

Note
跳过locked row 的query 会返回 不一致的 data view。 因此， skip locked 不适用于 一般事务性工作。 但是当 多个session 访问 同一个 queue-like table 时， 它可以用来 避免锁争用。

nowait 和 skip locked 仅适用于 row-level lock.

使用 nowit 或 skip locked 的语句 对于 基于 语句的 复制 是不安全的。

下面的例子 演示了 nowait 和 skip locked 。
session1 启动一个 对 单个 record 进行 row lock 的 事务，
session2 尝试使用 nowwait 对 同一条 record 进行 locking read。 因为 session1 lock 了 row，所以 返回错误
session3 使用 skip locked 进行 locking read 会读取 请求的row ，除了 session1 锁定的 row。

```
# Session 1:
mysql> CREATE TABLE t (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;
mysql> INSERT INTO t (i) VALUES(1),(2),(3);
mysql> START TRANSACTION;
mysql> SELECT * FROM t WHERE i = 2 FOR UPDATE;
+---+
| i |
+---+
| 2 |
+---+

# Session 2:
mysql> START TRANSACTION;
mysql> SELECT * FROM t WHERE i = 2 FOR UPDATE NOWAIT;
ERROR 3572 (HY000): Do not wait for lock.
```

```
# Session 3:
mysql> START TRANSACTION;
mysql> SELECT * FROM t FOR UPDATE SKIP LOCKED;
+---+
| i |
+---+
| 1 |
| 3 |
+---+
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-locks-set.html

15.7.3 Locks Set by Different SQL Statements in InnoDB

一个 locking read，(如) update，delete 语句 一般 会设置 record lock 到 处理SQL语句 时被扫描到的 所有 index record。 语句中 是否有 排除row 的 where条件 无关紧要。
。。应该是说：只要扫描到的 就会加 锁，即使 后来 被 where 排除了。
。。index record 直译 索引记录， 是index 还是 record? 是 被index 的 record?
。。不走index 就 锁全表？好像是，毕竟 没有index 的话，就只能 全表扫描了， 这样 全表的 row 都被加锁了，。。不过 表中所有的row 加锁 和 表锁 还是有区别的， 能不能 insert 的 区别。
。。。百度： lock record:
index record可以理解为索引行，MySQL B树索引的每一行内容是 索引键值+主键；
锁定读，update，delete通常都会在其扫描的索引行上加锁。
。。不太理解，那么怎么 锁row 呢？ 看上面的描述 是 lock index， 没有 lock row啊。
。。。

InnoDB 不 remember 确切的 where条件， 只知道 哪些index range 被 扫描到。
lock 一般是 next-key lock，所以 也会 阻塞 record 前面的 gap 中的 insert语句。
。。next-key lock 这个名字，像是 锁了 record 后面的 gap， 结果 是锁 前面的 gap。。

可以 显式禁用 gap lock， 这个会导致 不使用 next-key lock。
更多信息，看 15.7.1 ， InnoDB locking
事务隔离级别 也会 影响到 使用了什么锁，查看 15.7.2.1
。。15.7.1 中， next-key lock 会 送一个 (最大的值，无限大) 的 gap lock.

如果 搜索中 使用了 二级索引，且 要设置的 index record 是 exclusive(独占，互斥)，InnoDB 也会 检索 对应的 聚集index，并设置lock。
。。。ok了，所以 最终会锁 聚集索引
。。所以 InnoDB 必须有索引， 记得看到过：InnoDB 必须有主键。。就是 oneNote的 MySQL 页："InnoDB 是聚集索引，MyISAM 是非聚集索引。聚簇索引的文件存放在主键索引的叶子节点上，因此 InnoDB 必须要有主键"

如果 没有合适的 index 来执行SQL语句，且 MySQL 必须 扫描 整个表 来处理SQL语句，那么

表的 每行 都会被 lock，这个会 阻塞 其他用户的 所有 insert。
创建一个 好的index 是重要的，这样 你的 query 不会 scan 太多的row。

InnoDB 设置如下的 特殊的锁类型
select .. from
是一个 一致读(consistent read)，读取 数据库的 snapshot，且 不设置lock， 除非 事务隔离级别是 serializable，对于 serializable 级别，search 设置 shared next-key lock 到它遇到的 index record。但是，对于 使用唯一索引 锁定row 以搜索唯一row 的语句 只 需要一个 index record lock。

select .. for update, select .. for share
对于扫描到的row 使用 unique index acquire lock，and 释放 那些 不包含在 result set 中 的 row (如，如果这些row 不满足 where 条件)。
但是，有些情况下，row 不会被立刻 unlock，因为 在query执行期间，result row 和 它的source 之间的 关系 丢失了。例如，在 union中，从表中 扫描到的 且 lock的 row 可能会 在计算这些row 是否是result set 之前 被 插入到 临时表， 这种情况下， 临时表中的 row 和 原始表中的 row 的 关系 被丢失了，且 后者 不会 unlock，直到 query 执行完。

对于locking read (select + for update / for share), update, delete
根据 SQL 是 使用了 具有唯一搜索条件 的 unique index 还是 range类型的搜索条件 来决定 使用什么lock：
对于 具有唯一搜索条件的 唯一index，InnoDB 只锁定 找到的index，不会锁 index前面的 gap
对于其他搜索条件，和对于 非-unique 索引， InnoDB 锁 扫描到的 index range， 使用 gap lock 或 next-key lock 来 阻塞 其他session的 将数据插入到gap中的 insert。

对于 搜索时遇到的 index，select .. for update 语句会阻塞 其他 session 执行 select .. for share，或 阻塞其他session 在某些事务隔离级别 下的 reading。
一致读(consistent read) 会忽略 read view 上的 任何 lock。

update .. where
设置 独占 next-key lock 到 搜索时遇到的 所有 record。 但是，对于 使用 unique index 来搜索 unique row 的语句，只需要一个 index lock 来 lock row。

当 update语句 修改 聚簇index，会隐式 lock 受影响的 二级index。
当在插入二级index 之前 执行 双重check scan 时， 或 在插入新的 二级index 时， update 语句 会在 受影响的二级index 上 获得 share lock。

delete from .. where
设置 独占 next-key lock 到 搜索时遇到的 所有 记录上。但是，当 使用unique index 来搜索 unique row 时，只需要一个 index lock 来 lock row。

insert
设置 独占锁 在 被insert的row 上。这个lock 是 index-record lock，不是 next-key lock (即，这里no gap lock，不会阻塞在 被insert的row 前面的 gap 中的 insert 操作)。

在插入row之前，设置一个 插入意图间隙锁(insert intention gap lock) (这是一个gap

lock)。 这个lock 表明 插入的意图，即插入到同一个index gap 中的 多个事务，如果不在 间隙中的 同一个位置插入，则无需彼此等待。
假设 有值为4 和 7 的index，不同的事务 分别尝试 插入5 和6，每个事务都会 使用 insert intention gap lock 来 锁4-7的gap (在 获得 插入的row 的 排他锁 之前)， 但是 不会互相阻塞，因为 row 没有冲突。
如果 duplicate-key 的错误 发生，会 设置一个 共享锁 到 duplicate index record。
如果 多个 session 尝试 插入 相同的row (如果 另一个session 已经获得了 排他锁)，则 share lock 的使用 可以导致 deadlock。
如果另一个session删除了row，则死锁也是可以发生的。
。。这里有2个例子。

insert .. on duplicate key update
和 简单的 insert 不同，当 duplicate-key 错误发生时，会设置 独占锁 而不是共享锁到 row上(。。row上的锁最终还是体现到 index上)。 对于重复主键的值 采用 独占 index-record lock。 对于 重复唯一key的值 采用独占 next-key lock。

replace
如果在 unique key 上没有 碰撞，则 和 insert 一样。
否则，设置 独占 next-key lock 到 将被replace 的 row 上。

insert into T select .. from S where ..
设置 独占 index record lock (不带 gap lock) 到 插入到T 的每个 row 上。
如果 事务隔离级别 是 read committed， InnoDB 在S 上的 搜索 是一个 一致读 (consistent read) (no lock)。
否则(。。指不是 read committed)，InnoDB 设置 共享的next-key lock 在 从S 来的 每个row 上。
下面的情况，InnoDB 必须设置 lock ： 在使用基于语句的 binary log 进行 前滚(roll-forward) 恢复 期间，每条SQL语句 必须 以 和当初完全相同的 方式执行。

create table .. select .. 使用 共享next-key lock 来执行 select，或作为 一致读来执行 select，如 insert .. select。

当 使用 select 来构造 replace into T select .. from S where .. 或 update T .. where COL in (select .. from S ..)，在 从 S 来的 每个 row 上设置 共享的 next-key lock。

InnoDB 在初始化 表中 被定义为 AUTO_INCREMENT 的列时，在 和AUTO_INCREMENT列 关联的 index 的最后 设置一个 独占lock。
使用 innodb_autoinc_lock_mode=0, InnoDB 使用 特殊的 AUTO-INC table lock 模式，lock 被获取 并 hold 直到 当前SQL 语句 结束 (不是到 整个事务的 结束)，当 访问 auto-increment counter 时。
其他客户端 不会插入到table 中，只要 AUTO-INC table lock 被 hold。
innodb_autoinc_lock_mode=1 的 "bulk insert" 的 行为 和上面一样。
innodb_autoinc_lock_mode=2 ，不会使用 表级别的 AUTO-INC lock。

InnoDB 获得 之前初始化的 AUTO_INCREMENT 列 的值 不需要 设置 任何 lock。

如果在 表上 有 foreign key 的约束，所有 需要检查约束条件的 insert，update，delete

设置 共享的 record-level lock 到 record 上。 当约束失败时，InnoDB 也会设置 这些锁。

lock tables
    设置 表锁，但 它是 在 比 InnoDB 层 更高的 MySQL层 上设置这些锁。如果 innodb_table_locks=1 (默认) 和 autocommit=0 和 InnoDB 上的 MySQL层知道 row-level lock，则Innodb 会感知到 table lock。
    否则，InnoDB 的 死锁自动检测 不会 发现 涉及此类表锁的 死锁。而且，由于 高层的 MySQL层 不知道 row-level lock，它 可能对 另一个session正在加 row-level lock 的表 加 table lock。 当时，这不会 影响 事务完整性。

lock tables
    如果 innodb_table_locks=1(默认)，则 每个表 需要 2个lock。除了 MySQL层 的 table lock，它也 需要 一个 innodb table lock。要 避免 获得 InnoDB table lock，设置 innodb_table_locks=0。 如果 不需要 innodb table lock，那么 即使其他事务 lock 了表中的 一些record，lock tables 还是可以完成。
    在 MySQL 8.0中， innodb_table_locks = 0, 对于 显式的 lock tables .. write 的 table lock 无效。 对于 隐式(比如，通过trigger)的 lock tables .. write 或 lock tables .. read 的读写 的 table lock 会起效。

事务 hold 的所有 InnoDB 的lock 会在 事务 commit 或 abort 时 release。因此，在 autocommit=1的 InnoDB表 上调用 lock tables 没有太大的意义，因为 获得 InnoDB的 table lock 会被立刻 release。

你无法在 事务的一半时 lock 其他的表，因为 lock tables 执行 隐式 commit 和 unlock tables。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-next-key-locking.html
15.7.4 Phantom Rows

当 一个事务 在不同的时间点，执行相同的查询，但是 获得了 不同的 result set，那么 就称为 发生了 phantom problem。
例如，一个 select 执行了2次，但是 第二次返回的 result set 中 有一条row 是在 第一次返回的result set中不存在的，这个多出来的 row 就是 phantom row。

假设 child 表的 id列 有一个索引， 你想 read 和lock 所有 id 大于100 的 row，稍后你想更新这些 被选择的row 中的一些列：
    SELECT * FROM child WHERE id > 100 FOR UPDATE;

查询 从第一个id大于100的 record 开始 扫描index。 假设表 包含 id为90 和102 的row。如果 设置在index 上的 lock 不锁定 gap，其他session可以插入新的 row 到 table中，id是101，如果你 在同一个事务中 再次执行 select，你会看到 id为101的 phantom row。这违反了 事务的 隔离性。

为了防止 phantom， InnoDB 使用 next-key locking，它组合了 index-row locking 和 gap locking。
InnoDB 执行 row-level locking 的方式：当它 搜索或扫描 table index 时，它设置 共享

或独占 锁 到 它遇到的 index record 上。因此，row-level lock 实际上是 index-record lock。另外，index 上的 next-key lock 也会 影响 index 前面的gap。即，next-key lock 是 index-record lock 加 (index前面的gap上的) gap lock。 如果一个会话 在 索引的 记录 R 上有 共享或独占锁，另一个 会话 不能在 R前面的 gap 中插入 index record。

当 InnoDB 扫描 index时， 它也可以 锁 index 的最后一个 record 后的 gap。

你可以使用 next-key lock 来 实现一个 唯一性检查：如果 你 以共享模式 读取数据，且没有看到 你将插入的row 的 duplicate，那么你可以 安全地 插入你的 row， 你知道 在read 时， 设置在你的row 后面的记录上的 next-key lock 阻止 任何人插入 你的row 的 duplicate。 因此，next-key lock 使得你 "lock" 不存在的东西。

15.7.1 中提到 gap lock 可以被 禁用。这可能导致 phantom problem 由于 其他会话 可以插入 新的row 到 gap 中。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlocks.html
15.7.5 Deadlocks in InnoDB

死锁是不同的事务 由于它们hold 了其它事务需要的lock，所以导致它们都无法继续执行。它们都在等待资源可用，而不会release 它们已有的lock。

当事务在多张表中 以 相 反 的 顺 序 (通过 update 或 select .. for update 等语句) lock row 时，可能发生死锁。
锁定 index 和 gap 时，也可能死锁，因为每个事务 都获取了一些锁，但是由于 时间问题 而没有获取其他锁。

要降低死锁的概率，
使用事务 而不是 lock tables语句。
让 insert或update 的 事务 尽可能小，这样它们不会长时间存活。
当不同事务要更多 多张表 或 大范围的row 时，每个事务 使用 相同顺序的 操作。
对于被 select .. for update 和 update .. where 中用到的 列 创建 索引。

死锁的概率 不受 隔离级别的影响，因为 隔离级别 修改的是 read 操作， 而 死锁是由于 write引起的。

当 死锁侦测可用(默认是可用的)，且发生了 死锁， InnoDB 检查 某些条件，然后 回滚某个事务。
如果 死锁侦测不可用(通过 innodb_deallock_detect 禁用)，InnoDB 依赖 innodb_lock_wait_timeout 来回滚事务，如果有死锁的话。因此，即使你的应用逻辑是正确的，你也必须处理 事务的retry的情况。
。。默认50秒超时，如果我真的需要锁50秒呢？怎么区别 是真的需要50秒，还是 因为死锁导致50秒？
在 innodb 用户事务中，使用 show engine innodb status 查看 最后一个 死锁。
如果频繁死锁，使用 innodb_print_all_deadlocks 来 打印 所有死锁信息 到 mysqld 的 error日志中。(。。因为show engine innodb status 只能看到最后一个 死锁)

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-example.html
15.7.5.1 An InnoDB Deadlock Example

涉及2个客户端，A和B。

A 启用了 innodb_print_all_deadlocks，创建2张表:Animals 和 Birds，插入数据到这2张表。
A 启动事务，从 Animals 中以 share mode 选择一条row。

```
mysql> SET GLOBAL innodb_print_all_deadlocks = ON;
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE Animals (name VARCHAR(10) PRIMARY KEY, value INT) ENGINE =
InnoDB;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE Birds (name VARCHAR(10) PRIMARY KEY, value INT) ENGINE =
InnoDB;
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO Animals (name,value) VALUES ("Aardvark",10);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Birds (name,value) VALUES ("Buzzard",20);
Query OK, 1 row affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT value FROM Animals WHERE name='Aardvark' FOR SHARE;
+-------+
| value |
+-------+
|    10 |
+-------+
1 row in set (0.00 sec)
```

然后，B启动事务，以 share模式 在Birds 中选择一条 row

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT value FROM Birds WHERE name='Buzzard' FOR SHARE;
+-------+
| value |
+-------+
|    20 |
```

```
+-------+
1 row in set (0.00 sec)
```

Performance Schema 展示了 2条select语句后的 lock 情况

```
mysql> SELECT ENGINE_TRANSACTION_ID as Trx_Id,
              OBJECT_NAME as `Table`,
              INDEX_NAME as `Index`,
              LOCK_DATA as Data,
              LOCK_MODE as Mode,
              LOCK_STATUS as Status,
              LOCK_TYPE as Type
       FROM performance_schema.data_locks;
+-----------------+---------+---------+-----------+--------------+---------+-
-------+
| Trx_Id          | Table   | Index   | Data      | Mode         | Status  |
Type   |
+-----------------+---------+---------+-----------+--------------+---------+-
-------+
| 421291106147544 | Animals | NULL    | NULL      | IS           | GRANTED |
TABLE  |
| 421291106147544 | Animals | PRIMARY | 'Aardvark' | S,REC_NOT_GAP | GRANTED |
RECORD |
| 421291106148352 | Birds   | NULL    | NULL      | IS           | GRANTED |
TABLE  |
| 421291106148352 | Birds   | PRIMARY | 'Buzzard'  | S,REC_NOT_GAP | GRANTED |
RECORD |
+-----------------+---------+---------+-----------+--------------+---------+-
-------+
4 rows in set (0.00 sec)
```

B开始 update Animals 中的row

```
mysql> UPDATE Animals SET value=30 WHERE name='Aardvark';
```

B 必须等待。Performance Schema 展示了 等待lock：

```
mysql> SELECT REQUESTING_ENGINE_LOCK_ID as Req_Lock_Id,
              REQUESTING_ENGINE_TRANSACTION_ID as Req_Trx_Id,
              BLOCKING_ENGINE_LOCK_ID as Blk_Lock_Id,
              BLOCKING_ENGINE_TRANSACTION_ID as Blk_Trx_Id
       FROM performance_schema.data_lock_waits;
+-------------------------------------+-----------+----------------------
--------------+-----------------+
| Req_Lock_Id                         | Req_Trx_Id | Blk_Lock_Id
| Blk_Trx_Id      |
+-------------------------------------+-----------+----------------------
--------------+-----------------+
| 139816129437696:27:4:2:139816016601240 |      43260 |
139816129436888:27:4:2:139816016594720 | 421291106147544 |
```

```
+----------------------------------------+-----------+-----------------------
--------------+----------------+
1 row in set (0.00 sec)

mysql> SELECT ENGINE_LOCK_ID as Lock_Id,
              ENGINE_TRANSACTION_ID as Trx_id,
              OBJECT_NAME as `Table`,
              INDEX_NAME as `Index`,
              LOCK_DATA as Data,
              LOCK_MODE as Mode,
              LOCK_STATUS as Status,
              LOCK_TYPE as Type
       FROM performance_schema.data_locks;
+----------------------------------------+----------------+---------+---------
+-----------+--------------+---------+-------+
| Lock_Id                                | Trx_Id         | Table   | Index
| Data      | Mode         | Status  | Type  |
+----------------------------------------+----------------+---------+---------
+-----------+--------------+---------+-------+
| 139816129437696:1187:139816016603896   |          43260 | Animals | NULL
| NULL      | IX           | GRANTED | TABLE |
| 139816129437696:1188:139816016603808   |          43260 | Birds   | NULL
| NULL      | IS           | GRANTED | TABLE |
| 139816129437696:28:4:2:139816016600896 |          43260 | Birds   | PRIMARY
| 'Buzzard' | S,REC_NOT_GAP | GRANTED | RECORD |
| 139816129437696:27:4:2:139816016601240 |          43260 | Animals | PRIMARY
| 'Aardvark' | X,REC_NOT_GAP | WAITING | RECORD |
| 139816129436888:1187:139816016597712   | 421291106147544 | Animals | NULL
| NULL      | IS           | GRANTED | TABLE |
| 139816129436888:27:4:2:139816016594720 | 421291106147544 | Animals | PRIMARY
| 'Aardvark' | S,REC_NOT_GAP | GRANTED | RECORD |
+----------------------------------------+----------------+---------+---------
+-----------+--------------+---------+-------+
6 rows in set (0.00 sec)
```

当事务 尝试修改数据库时，InnoDB 只使用 sequential transaction id， 所以 上面的 只读事务id 从421291106148352 变成 43260.

如果此时，A尝试更新 Birds 中的row， 那么会导致 死锁

```
mysql> UPDATE Birds SET value=40 WHERE name='Buzzard';
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting
transaction
```

InnoDB 回滚导致死锁的 事务。来自B 的第一个 update，可以执行。

Information Schema 包含了 死锁的次数

```
mysql> SELECT `count` FROM INFORMATION_SCHEMA.INNODB_METRICS
          WHERE NAME="lock_deadlocks";
```

```
+-------+
| count |
+-------+
|     1 |
+-------+
1 row in set (0.00 sec)
```

InnoDB status 包含了下面的 关于死锁和事务 的 信息。也展示了 read-only transaction id 421291106147544 变成了 sequential transaction id 43261

```
mysql> SHOW ENGINE INNODB STATUS;
------------------------
LATEST DETECTED DEADLOCK
------------------------
2022-11-25 15:58:22 139815661168384
*** (1) TRANSACTION:
TRANSACTION 43260, ACTIVE 186 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)
MySQL thread id 19, OS thread handle 139815619204864, query id 143 localhost u2
updating
UPDATE Animals SET value=30 WHERE name='Aardvark'

*** (1) HOLDS THE LOCK(S):
RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.
`Birds` trx id 43260 lock mode S locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 7; hex 42757a7a617264; asc Buzzard;;
 1: len 6; hex 00000000a8fb; asc       ;;
 2: len 7; hex 82000000e40110; asc        ;;
 3: len 4; hex 80000014; asc     ;;


*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.
`Animals` trx id 43260 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 8; hex 416172647661726b; asc Aardvark;;
 1: len 6; hex 00000000a8f9; asc       ;;
 2: len 7; hex 82000000e20110; asc        ;;
 3: len 4; hex 8000000a; asc     ;;


*** (2) TRANSACTION:
TRANSACTION 43261, ACTIVE 209 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)
MySQL thread id 18, OS thread handle 139815618148096, query id 146 localhost u1
updating
UPDATE Birds SET value=40 WHERE name='Buzzard'
```

```
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.
`Animals` trx id 43261 lock mode S locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 8; hex 416172647661726b; asc Aardvark;;
 1: len 6; hex 00000000a8f9; asc       ;;
 2: len 7; hex 82000000e20110; asc        ;;
 3: len 4; hex 8000000a; asc     ;;


*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.
`Birds` trx id 43261 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 7; hex 42757a7a617264; asc Buzzard;;
 1: len 6; hex 00000000a8fb; asc       ;;
 2: len 7; hex 82000000e40110; asc        ;;
 3: len 4; hex 80000014; asc     ;;

*** WE ROLL BACK TRANSACTION (2)
------------
TRANSACTIONS
------------
Trx id counter 43262
Purge done for trx's n:o < 43256 undo n:o < 0 state: running but idle
History list length 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 421291106147544, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 421291106146736, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 421291106145928, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 43260, ACTIVE 219 sec
4 lock struct(s), heap size 1128, 2 row lock(s), undo log entries 1
MySQL thread id 19, OS thread handle 139815619204864, query id 143 localhost u2
```

error 日志包含了 事务和lock 的信息

```
mysql> SELECT @@log_error;
+--------------------+
| @@log_error        |
+--------------------+
| /var/log/mysqld.log |
+--------------------+
1 row in set (0.00 sec)

TRANSACTION 43260, ACTIVE 186 sec starting index read
mysql tables in use 1, locked 1
```

```
LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)
MySQL thread id 19, OS thread handle 139815619204864, query id 143 localhost u2
updating
UPDATE Animals SET value=30 WHERE name='Aardvark'
RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.
`Birds` trx id 43260 lock mode S locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 7; hex 42757a7a617264; asc Buzzard;;
 1: len 6; hex 00000000a8fb; asc       ;;
 2: len 7; hex 82000000e40110; asc        ;;
 3: len 4; hex 80000014; asc     ;;

RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.
`Animals` trx id 43260 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 8; hex 416172647661726b; asc Aardvark;;
 1: len 6; hex 00000000a8f9; asc       ;;
 2: len 7; hex 82000000e20110; asc        ;;
 3: len 4; hex 8000000a; asc     ;;

TRANSACTION 43261, ACTIVE 209 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1128, 2 row lock(s)
MySQL thread id 18, OS thread handle 139815618148096, query id 146 localhost u1
updating
UPDATE Birds SET value=40 WHERE name='Buzzard'
RECORD LOCKS space id 27 page no 4 n bits 72 index PRIMARY of table `test`.
`Animals` trx id 43261 lock mode S locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 8; hex 416172647661726b; asc Aardvark;;
 1: len 6; hex 00000000a8f9; asc       ;;
 2: len 7; hex 82000000e20110; asc        ;;
 3: len 4; hex 8000000a; asc     ;;

RECORD LOCKS space id 28 page no 4 n bits 72 index PRIMARY of table `test`.
`Birds` trx id 43261 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 7; hex 42757a7a617264; asc Buzzard;;
 1: len 6; hex 00000000a8fb; asc       ;;
 2: len 7; hex 82000000e40110; asc        ;;
 3: len 4; hex 80000014; asc     ;;
```

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-detection.html
15.7.5.2 Deadlock Detection

当 死锁侦测 被启用(默认启用)，Innodb 自动 侦测 事务死锁，并 回滚 一个或多个事务 来打破 死锁。 InnoDB 尝试 选择 回滚 小的事务，事务的大小由 被insert，update，delete的row 的行数来决定。

如果 innodb_table_locks=1(默认) 且 autocommit=0 且 InnoDB之上的MySQL层 知道 row-level lock， 那么 Innodb 就知道 table lock。 否则 InnoDB 无法 检测到 MySQL的lock tables 锁定的table 引起的死锁， 或 其他存储引擎设置的 lock 引起的死锁， 要解决这些死锁，需要使用 innodb_lock_wait_timeout。

如果 InnoDB Monitor 的输出 的 LATEST DETECTED DEADLOCK 章节 包含了：TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH, WE WILL ROLL BACK FOLLOWING TRANSACTION， 这表明 在等待列表中的 事务 数量 已经达到了 200。等待列表中的 事务超过 200 被认为 是死锁， 尝试 check 等待列表的 事务 被回滚。 相同的错误 也会发生，如果 locking thread 必须查看 等待列表中 事务拥有的 超过 100万个锁时。

禁用死锁侦测
在 高并发系统，死锁侦测 可能导致 slowdown，当很多线程 等待 同一个锁时。
此时，禁用死锁侦测，依靠 innodb_lock_wait_timeout 来 进行回滚事务 可能更 高效。
通过 innodb_deadlock_detect 来禁用。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlocks-handling.html
15.7.5.3 How to Minimize and Handle Deadlocks

如何管理数据库操作 来最小化死锁。 应用中 需要的 error handling。

死锁 在 事务性数据库中 是一个 经典的问题， 但 通常没有危害，除非 频率很高，导致你无法 执行正常的 事务。
通常，你必须 在代码中 编写 由于死锁而导致的回滚 后的 处理逻辑。

InnoDB 使用 自动的 row-level locking。你可能 在 insert一行 或delete一行时 就 遇到死锁，这是因为这些操作 不是 原子性的， 它们自动 在 insert或delete 的 row 的 (可能多个) index 记录上 设置 锁。

你可以应对死锁，降低它们发生的概率，通过以下技术
使用 show engine innodb status 来 确定 最近的死锁的 原因。 这可以帮助你 调整应用 以避免死锁。

如果 频繁死锁， 启用 innodb_print_all_deadlocks 来 收集 更多 信息。 当你完成debug 的时候，关闭这个 选项。

始终准备：由于死锁导致事务失败时 的重试。 死锁不是 危害性的。 just try again。

让 事务小 且 持续时间短 来 减小发生冲突的可能性。

在执行完 一组相关更改 后立即commit，来减少冲突。特别是，不要让 interactive mysql session 由于 未提交的事务 而 长时间 打开。

如果你使用 locking read (select .. for update/share)， 尝试使用 低级的 隔离级别，比如 read committed

当 在一个事务中修改多个表，或 一个表中不同的row集合，每次都以 一致的顺序 执行。那么事务会 形成良好的队列，不会死锁。 例如，将 数据库操作 放到 function 中，或者 调用 stored routine，而不是 多个 insert，uodpate，delete 的组合。
。。数据库的函数， 数据库的存储过程。

创建 精选的index，来使得 你的query scan 更少的index记录，设置更少的lock。 使用 explain select 来确定 MySQL server 认为 哪些index 最适合你的 query。

使用更少的locking，如果 可以接受 select 从old snapshot 返回 数据，那么不要 增加 for update,for share。 此时 使用 read committed 隔离级别 是good， 因为 同一个事务中的 每个 consistent read 从 它自己的 fresh snapshot 中读。
。。
普通读（也称一致性读，英文名：Consistent Read）。
这个就是指普通的SELECT语句，在末尾不加FOR UPDATE或者LOCK IN SHARE MODE的SELECT语句。普通读的执行方式是生成ReadView直接利用MVCC机制来进行读取，并不会对记录进行加锁。
。。

如果没有其他的办法有效，那么 使用 table-level lock 序列化 事务。正确的 对 transactional table(比如InnoDB的表) 使用 lock tables 的方式是： 使用 set autocommit=0 (而不是 start transaction) 来启动 事务，然后 lock tables， 不要调用 unlock tables 直到 你显式 commit掉事务。 例如，你想要 对表t1写，对表t2读，你可以：

```
SET autocommit=0;
LOCK TABLES t1 WRITE, t2 READ, ...;
... do something with tables t1 and t2 here ...
COMMIT;
UNLOCK TABLES;
```

表级锁阻止 对表的并发update，避免死锁，代价是系统响应能力。

。。set autocommit=0指事务非自动提交,自此句执行以后,每个SQL语句或者语句块所在的事务都需要显示"commit"才能提交事务

另一个序列化 事务 的方式是 创建一个 辅助的 "semaphore" 表，只包含一行数据。每个 事务 在 访问其他表之前 都需要 更新那行。这样，所有的 事务 都以 序列化的顺序 发生。
注意，innodb 的 死锁侦测 也会起效，因为 序列化用的锁 是 row-level lock。对于 MySQL 表级锁 ，必须使用 超时 来解决死锁。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-scheduling.html

15.7.6 Transaction Scheduling
事务调度

InnoDB 使用 Contention-Aware Transaction Scheduling (CATS) (竞争感知的事务调度) 算法来 对 确定 正在等待lock的事务的优先级。 当 多个事务 在等待 同一个对象的 lock，CATS 决定 哪个事务 先获得锁。
。。。所以 任何锁竞争 都可以用这个 算法？ 来确定 下一个是谁获得锁，而不是 再次竞争？

CATS算法 对 等待中的 事务进行 优先级排序 by 赋予一个 调度权重，这个是 基于 被这个事务 阻塞的 事务的 数量 来计算的。 例如，如果 2个 事务在等待 一个lock， 阻塞了更多事务的 事务 被赋予 更大的权重。 如果 权重相同， 等待时间长的 事务 优先。
。。就是 有多少事务 在等待 本事务已经lock的资源 (等于就是 其他事务 被 本事务阻塞了，需要本事务 执行完，释放lock，其他事务 才能 继续执行下去)。

注意：
在8.0.20之前， InnoDB 也使用 FIFO 算法 来 调度 事务，只有在 锁竞争 激烈的时候 使用 CATS。 8.0.20，对CATS 进行了增强，使得 FIFO 被淘汰。 事务调度算法的修改 可能影响 事务被授予锁 的顺序。

你可以 查询 Information Schema 的 Innodb_trx 表 来查看 事务调度权重。 只会为 等待中的 事务 计算 权重。 等待中的事务 是指 transaction execution state 为 LOCK WAIT 的事务， 被展现在 trx_state 列。 现在没有等待锁 的 事务的 trx_schedule_weight 值为 NULL。

INNODB_METRICS counter 被用来 监控 code-level transaction scheduling event。查看 15.15.6 获得 innodb_metircs 的信息。

- lock_rec_release_attemptes
  释放 record lock 的 尝试次数。 一个尝试 可能导致 0或更多 lock 被 释放，因为 单个结构中 可能有 0个或多个 记录锁。
- lock_rec_grant_attempts
  授予(grant) record lock 的尝试次数。 一个尝试 可能导致 0或多个 record lock 被 grant
- lock_schedule_refreshes
  分析 锁等待(依赖关系拓扑)图来更新 调度事务的权重 的次数。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-configuration.html
15.8 InnoDB Configuration

15.8.1 InnoDB Startup Configuration
15.8.2 Configuring InnoDB for Read-Only Operation
15.8.3 InnoDB Buffer Pool Configuration
15.8.4 Configuring Thread Concurrency for InnoDB
15.8.5 Configuring the Number of Background InnoDB I/O Threads

15.8.6 Using Asynchronous I/O on Linux
15.8.7 Configuring InnoDB I/O Capacity
15.8.8 Configuring Spin Lock Polling
15.8.9 Purge Configuration
15.8.10 Configuring Optimizer Statistics for InnoDB
15.8.11 Configuring the Merge Threshold for Index Pages
15.8.12 Enabling Automatic Configuration for a Dedicated MySQL Server

启动的配置
对于只读操作的配置
缓冲池
线程并发
后台IO线程数量
Linux上使用 异步IO
IO 容量
旋转锁定轮询 spin lock polling
purge(净化，清除，清洗) 配置
优化器统计信息
index 页 的merge
专业MySQL server的自动配置。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-init-startup-configuration.html
15.8.1 InnoDB Startup Configuration

涉及 data文件，log文件，page size，memory buffer 的配置， 这些 应该在 初始化 InnoDB 前 确定。 在初始化后 修改这些配置 可能 需要一些额外的处理。

Specifying Options in a MySQL Option File
由于MySQL 使用 data file，log file,page size 的配置 来初始化 InnoDB，所以 建议你 定义这些 配置 到 option file，MYSQL 在 启动是会读取文件 并初始化 InnoDB。 通常，InnoDB 在 MySQL server 第一次启动时 初始化。

你可以放置 InnoDB配置 到 文件的 [mysqld] 组。

确保 mysqld 从 指定文件中 (和mysqld-auto.cnf) 读取配置，启动时，使用 --defaults-file 作为 命令行的 第一个参数。

```
mysqld --defaults-file=path_to_option_file
```

Viewing InnoDB Initialization Information
查看启动期间 的 InnoDB 初始化信息。
window：

```
C:\> "C:\Program Files\MySQL\MySQL Server 8.0\bin\mysqld" --console
```

Linux：

```
$> bin/mysqld --user=mysql &
```

如果你没有 发送 server output 到 console，在启动后 检查error log 来查看 启动期间的

InnoDB 初始化信息。

Important Storage Considerations
重要的存储考虑事项
在进行启动配置前 考虑下面的 存储相关的 事项。

有些情况下，你可以 提高数据库性能 by 将data 和 log 文件 放到不同的物理磁盘。你可以 为 InnoDB data 文件 使用 原始磁盘分区(原始设备)(raw disk partitions(raw devices)) 来提高 IO速度。

InnoDB 是一个 使用commit，rollback的 事务安全的(符合ACID 的) 存储引擎，有崩溃恢复 能力 来保护用户数据。
但是 如果底层的OS 或硬件 无法预期地工作，那么InnoDB 是没有办法做到前面提到的事项的。
许多OS 或 disk子系统 可能 delay 或 re-order 写操作 来提升 性能。
在一些OS上，fsync() 应该等待 文件的所有未写入数据 都被刷新，但是 实际上 可能在数据被刷新到 稳定存储 前就返回。
因此，OS崩溃 或 停电 可能破坏 最近committed 数据，最坏情况下，会损坏数据库，因为 写操作被 重排序了。
如果 数据完整性 很重要，在上生产前，执行 "pull-the-plug" 测试(。。拔插头，但是不能保证下次可以用啊，而且现在也没有写数据)。在MacOS，InnoDB 使用 fcntl() 文件刷新方法。 在Linux，建议禁用 write-back cache
在 ATA/SATA 磁盘上，命令 "hdparm -W0 /dev/hda" 可能可以 禁用 write-back cache。记住，有些设备 或 disk 控制器 可能无法 禁用 write-back cache

关于 InnoDB 恢复能力，InnoDB 使用文件 flush技术，包括 称为 doublewrite buffer 的结构，它默认启用(innodb_doublewrite=ON)。 doublewrite buffer 增加了安全性 从意外的error或断点 中恢复，在大部分Unix 上 通过降低fsync()的调用次数 来 提升性能。如果你关注 数据完整性 或 错误恢复， 建议开启doublewrite。更多信息 15.11.1

在将 NFS 与 InnoDB 一起使用前，查看列出的问题：
https://dev.mysql.com/doc/refman/8.0/en/disk-issues.html#disk-issues-nfs

System Tablespace Data File Configuration
innodb_data_file_path 选项 定义了 Innodb 系统表空间数据文件 的 name，size，属性。如果你没有配置，那么默认是创建 single auto-extending data file，比12MB略大，文件名是 ibdata1:

```
mysql> SHOW VARIABLES LIKE 'innodb_data_file_path';
+-----------------------+------------------------+
| Variable_name         | Value                  |
+-----------------------+------------------------+
| innodb_data_file_path | ibdata1:12M:autoextend |
+-----------------------+------------------------+
```

value的格式：file_name:file_size[:autoextend[:max:max_file_size]]
文件size可以 用 K M G 来表示 kb，mb，g。 用K 的话必须是 1024的倍数，否则会 四舍五入到 最近的 M。 file size 的 sum 必须 大于12MB

你可以指定 多个 data file， 通过 分号分隔的列表

```
[mysqld]
innodb_data_file_path=ibdata1:50M;ibdata2:50M:autoextend
```

autoextend 和 max 属性 只能在 列表的 最后一个 data file 上用

指定 autoextend 时， data file 在需要时，自动增加 64MB， 这个值 被 innodb_autoextend_increment 控制。

在autoextend 属性后面 增加max属性 来 指定 auto-extending data file 的最大值。

```
[mysqld]
innodb_data_file_path=ibdata1:12M:autoextend:max:500M
```

第一个系统 表空间 数据文件强制执行 最小文件大小，以确保 有足够的空间 用于 doublewrite buffer page。 下面是对每种 Innodb page size 的最小 file size， 默认的 innodb page size 是 16384(16KB)。

Page Size (innodb_page_size)	Minimum File Size
16384 (16KB) or less	3MB
32768 (32KB)	6MB
65536 (64KB)	12MB

如果磁盘满了，你可以 在另一个 磁盘上 增加一个 data file。

单个文件的 size limit 由 OS 决定。只要OS支持，你可以设置大于4GB 的file size。
你可以使用 raw disk partitions 作为 data file。

InnoDB 不知道 文件系统 最大的 file size， 所以在 那些 最大file size 较小的 OS 上要注意。

系统 表空间 文件 默认 被创建在 datadir。 通过 innodb_data_home_dir 来指定位置。

```
[mysqld]
innodb_data_home_dir = /myibdata/
innodb_data_file_path=ibdata1:50M:autoextend
```

innodb_data_home_dir，值 必须以 / 结尾。
innodb不会创建目录，所以确保 指定的路径 存在。确保mysql 有权限创建文件。

可以指定 系统 表空间 数据 文件的 绝对路径，下面的路径 和上面的一样

```
[mysqld]
innodb_data_file_path=/myibdata/ibdata1:50M:autoextend
```

当 innodb_data_file_path 是绝对路径时，它不会和 innodb_data_home_dir 进行连接。

InnoDB Doublewrite Buffer File Configuration
从8.0.20开始， doublewrite buffer 存储区域 在 doublewrite file中，这为 doublewrite page的 存储位置 提供了 灵活性。
innodb_doublewrite_dir 定义了 创建 doublewrite 文件的 目录。如果没有指定，则创建到 innodb_data_home_dir 目录。

```
innodb_doublewrite_dir=/path/to/doublewrite_directory
```

doublewrite buffer 还有其它配置：doublewrite file的数量，每个线程的 page 的数量，doublewrite batch size。 更多信息 看 15.6.4

Redo Log Configuration
从8.3.30开始，redo log 占用的 磁盘空间 被 innodb_redo_log_capacity 控制。

```
[mysqld]
innodb_redo_log_capacity = 8589934592
```

innodb_redo_log_capacity 取代了 innodb_log_file_size 和 innodb_log_files_in_group，后2个已经 deprecated。如果定义了 innodb_redo_log_capacity，则 另两个 会被 忽略，否则， 用 file size * in group 的积 作为 log_capacity 的值。 如果都没有定义，则默认值100MB。 最大值是 128GB。

从8.0.30开始，innodb 尝试维护 32个 redo log file，每个文件 等于 1 / 32 * innodb_redo_log_capacity。

8.0.30之前，innodb 创建 2个 5mb 的 redo 文件，分别叫 ib_logfile0，ib_logfile1。你可以定义 redo文件的 数量 和 每个文件的大小，通过 innodb_log_file_size (默认48MB)，innodb_log_files_in_group

innodb_log_group_home_dir 定义了 log 文件的目录。你可以用来 设置 redo log文件 到 另一个 物理存储地址，来避免 IO资源 冲突

```
[mysqld]
innodb_log_group_home_dir = /dr3/iblogs
```

Undo Tablespace Configuration
undo log，默认下，位于 MySQL 初始化时 创建的 2个 undo tablespace 中。
innodb_undo_directory 定义了 innodb应该在哪里 创建 undo tablespace。 默认是在 data 目录中。
undo log 的IO模式 使得 undo tablespace 适合 SSD 存储
更多信息看 15.6.3.4

Global Temporary Tablespace Configuration
全局 临时 表空间 存储了 对用户创建的临时表所做更改的 rollback segment
默认是 innodb_data_home_dir 目录下 一个 名为 ibtmp1 的 初始时 略大于12MB 的文件。
innodb_temp_data_file_path 指定 全局临时表空间 data file 的 path,文件名，file size。

Session Temporary Tablespace Configuration
在8.0.15 及 之前， session 临时 表空间 存储 用户创建的临时表 和 (当 innodb 被配置为在磁盘上存储 内部临时表 (internal_tmp_disk_storage_engine=InnoDB) 时) 优化器创建的内部临时表。
8.0.16开始，innodb 总是 被用作 磁盘存储引擎 来存储 内部临时表。

innodb_temp_tablespaces_dir 定义了 innodb 创建的 session临时表空间 的目录。默认是 data 目录中的 #innodb_temp 目录

Page Size Configuration
innodb_page_size 指定了 MySQL实例中 所有 innodb表空间的 page size。
在初始化时设置，并且不能修改。
可用的选项有 64KB，32KB，16KB(默认)，8KB，4KB。 也可以用 字节数来指定(65536, 32768, 16384, 8192, 4096)
较小的 page size 对于 包含许多 小写入的 OLTP 更有效，因为 当单个page 包含多个 row 时，冲突会比较严重。
较小的 page size 对于 SSD 存储设备 也更有效，因为 通常使用了 small block size。
保持 innodb page size 接近 存储设置的 block size，来 最小化 回写disk时 的 未修改数据。

Memory Configuration
MySQL 为 各种提高 数据库操作的 性能的 cache和buffer 分配 内存。
为innodb 分配内存时，需要考虑 OS需要的内存，其他应用需要的内存,MySQL的其他类型的 buffer 和cache。例如，如果你使用 MyISAM 表，考虑 为 key buffer 分配的内存 (key_buffer_size)。 看8.12.3.1 获得 MySQL buffer 和cache 的总览

InnoDB 的buffer:
innodb_buffer_pool_size
定义了 buffer pool 的size，buffer pool 是 为innodb 的表 + index + 其他辅助 buffer 保存 cached data 的 内存区域。
size 对于 系统性能是重要的，通常建议 配置为 50% - 75% 的 系统内存(system memory)。
默认 buffer pool size 是 128mb。
查看 8.12.3.1 ： how mysql use memory

如果系统的内存很大，你可以提升 并发，通过 将 buffer pool 分割成 多个 buffer pool 实例。 buffer pool instance 的数量 通过 innodb_buffer_pool_instance 来控制。 默认 只创建一个 实例。

innodb_log_buffer_size
定义 innodb 在 写log 文件到 disk 时 使用的buffer 的size。
默认 16mb
大的log buffer 允许 大型事务 在 commit 之前 不需要 写log到磁盘。
如果 你有 update,insert,delete 许多row 的事务，你可以考虑 增加 log buffer 的 size 来 节约 磁盘IO

Linux上，如果 kernel 支持 large page，innodb可以用 large page 来 为 它的buffer pool 分配内存。。8.12.3.2，enabling large page support

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-read-only-instance.html
15.8.2 Configuring InnoDB for Read-Only Operation
你可以 通过 在server startup 时 激活 --innodb-read-only 配置选项 来 查询 mysql data 目录 是在 只读介质(read-only media) 上的 表。

innodb_change_buffering=0 启动server 并 正常停止server，来让 buffer里的数据 都刷新到 data file.

How to Enable
    --innodb-read-only=1
    --pid-file=path_on_writable_media
    --event-scheduler=disabled
    --innodb-temp-data-file-path
    innodb_read_only

Usage Scenarios
在只读存储介质(如DVD或CD) 上 分发mysql应用程序或一组mysql数据

多个mysql实例 同时查询 同一个 data directory，通常是在 data warehousing configuration 中。 你可以使用这个技术 (。。应该是指 多个instance 查询同一个 data directory。但不太对劲吧。IO还是共用的) 来 避免 在高负载下，mysql实例发生 瓶颈。或者你可以为 不同的实例 使用不同的配置，来 针对 特定类型的 查询 调整 每个实例。

查询 那些 由于安全 或数据完整性 的原因 进入只读状态的数据，如 备份的存档数据。

注意：这个特性 主要是为了 实现 分发 和 部署的灵活性，不是 为了 只读带来的 性能。对于只读查询，看8.5.3：optimizing innodb read-only transaction 来获得 性能调优， 这不会使得 整个server 变成 只读。

How It Works
没有 change buffering
启动时 没有 crash recovery 阶段
没有redo log，所以可以设置 innodb_log_file_size 为最小size (1MB)
大部分后台线程 停止。
死锁，监视器output 等 的信息 不会 写到 临时文件。
用于隔离级别的 mvcc 被关闭。
不使用 undo log， 可以禁用 innodb_undo_tablespaces 和 innodb_undo_directory

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-buffer-pool.html
15.8.3 InnoDB Buffer Pool Configuration

15.8.3.1 Configuring InnoDB Buffer Pool Size

15.8.3.2 Configuring Multiple Buffer Pool Instances
15.8.3.3 Making the Buffer Pool Scan Resistant
15.8.3.4 Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)
15.8.3.5 Configuring Buffer Pool Flushing
15.8.3.6 Saving and Restoring the Buffer Pool State
15.8.3.7 Excluding Buffer Pool Pages from Core Files

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-resize.html
15.8.3.1 Configuring InnoDB Buffer Pool Size

本章介绍的是 可以离线 和 服务器运行时， 都可以 进行的 Innodb buffer pool size 的配置。 更多 在线配置 buffer pool size，查看 https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-resize.html#innodb-buffer-pool-online-resize

当增加或减少 innodb_buffer_pool_size 时，操作按块执行(op is performed in chunch)。chunk size 通过 innodb_buffer_pool_chunk_size 定义，默认128M。更多信息：https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-resize.html#innodb-buffer-pool-chunk-size

buffer pool size 必须 是 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances 的 倍数(*1,*2..)。 如果 你配置的 innodb_buffer_pool_size 不是 上述 积 的倍数，buffer pool size 会自动 调整为 上述积的 倍数。
。。应该是 向上取倍数

在下面的例子中， innodb_buffer_pool_size 是 8g， innodb_buffer_pool_instances 是 16。innodb_buffer_pool_chunk_size 是默认的 128M
8g 是有效的， 因为是 16*128M=2g 的倍数。

```
$> mysqld --innodb-buffer-pool-size=8G --innodb-buffer-pool-instances=16

mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+----------------------------------------+
| @@innodb_buffer_pool_size/1024/1024/1024 |
+----------------------------------------+
|                           8.000000000000 |
+----------------------------------------+
```

下面的例子中，9g，16,128M， 9g 不是 16*128M 的倍数，所以 被调整为 10g。

```
$> mysqld --innodb-buffer-pool-size=9G --innodb-buffer-pool-instances=16

mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+----------------------------------------+
| @@innodb_buffer_pool_size/1024/1024/1024 |
+----------------------------------------+
```

```
|                                      10.000000000000 |
+--------------------------------------------+
```

Configuring InnoDB Buffer Pool Chunk Size
innodb_buffer_pool_chunk_size 可以 以 1mb(1048576 byte) 为最小单位 进行 增加 或减少。
只能在 启动时，通过 命令行 或 mysql配置文件 进行修改。

```
$> mysqld --innodb-buffer-pool-chunk-size=134217728

[mysqld]
innodb_buffer_pool_chunk_size=134217728
```

当改变 innodb_buffer_pool_chunk_size时，下面的 条件会被应用：
如果 在启动时， 新的 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances 大于 当前的 buffer pool size， innodb_buffer_pool_chunk_size 被缩小为 innodb_buffer_pool_size / innodb_buffer_pool_instances。
。。。？那这个 调整 和 上面的 innodb_buffer_pool_size 不是 倍数 的调整 那个先发生？
。。感觉是这个， 这个是 启动时， 但是 上面也是 mysqld 命令行，也是 启动时。
。。这个 在启动时发生， 如果后续 修改了 buffer pool，则 会调整 buffer pool， mysqld 不一定是启动时， 是 运行时调整 参数。

例如，如果 buffer pool 被初始化为 2g， 4 buffer pool instances， chunk size 是 1g，那么 chunk size 会变为 innodb_buffer_pool_size / innodb_buffer_pool_instances ：

```
$> mysqld --innodb-buffer-pool-size=2147483648 --innodb-buffer-pool-instances=4
--innodb-buffer-pool-chunk-size=1073741824;

mysql> SELECT @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|                2147483648 |
+---------------------------+

mysql> SELECT @@innodb_buffer_pool_instances;
+-------------------------------+
| @@innodb_buffer_pool_instances |
+-------------------------------+
|                             4 |
+-------------------------------+

# Chunk size was set to 1GB (1073741824 bytes) on startup but was
# truncated to innodb_buffer_pool_size / innodb_buffer_pool_instances

mysql> SELECT @@innodb_buffer_pool_chunk_size;
+--------------------------------+
| @@innodb_buffer_pool_chunk_size |
+--------------------------------+
|                      536870912 |
```

```
+--------------------------------+
```

buffer pool size 必须是 innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances 的 倍数，如果 调整 chunk size， 那么 innodb_buffer_pool_size 会自动调整为 倍数。
这个调整 发生在 buffer pool 初始化时。

```
# The buffer pool has a default size of 2GB (2147483648 bytes)

mysql> SELECT @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|                2147483648 |
+---------------------------+

# The chunk size is .5 GB (536870912 bytes)

mysql> SELECT @@innodb_buffer_pool_chunk_size;
+--------------------------------+
| @@innodb_buffer_pool_chunk_size |
+--------------------------------+
|                      536870912 |
+--------------------------------+

# There are 4 buffer pool instances

mysql> SELECT @@innodb_buffer_pool_instances;
+-------------------------------+
| @@innodb_buffer_pool_instances |
+-------------------------------+
|                             4 |
+-------------------------------+

# Chunk size is decreased by 1MB (1048576 bytes) at startup
# (536870912 - 1048576 = 535822336):

$> mysqld --innodb-buffer-pool-chunk-size=535822336

mysql> SELECT @@innodb_buffer_pool_chunk_size;
+--------------------------------+
| @@innodb_buffer_pool_chunk_size |
+--------------------------------+
|                      535822336 |
+--------------------------------+

# Buffer pool size increases from 2147483648 to 4286578688
# Buffer pool size is automatically adjusted to a value that is equal to
# or a multiple of innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances
```

```
mysql> SELECT @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|                4286578688 |
+---------------------------+
```

调整 chunk size时，小心， 因为 这个值的修改 会 增加 buffer pool 的size。

为了 避免潜在的性能问题， chunk 的数量 (innodb_buffer_pool_size / innodb_buffer_pool_chunk_size) 不应该 超过 1000

Configuring InnoDB Buffer Pool Size Online
innodb_buffer_pool_size 的值可以通过 set 语句 动态设置， 不需要重启服务器。

```
mysql> SET GLOBAL innodb_buffer_pool_size=402653184;
```

会等待所有 激活的事务完成后，才开始 resize。 resize的时候，需要使用 buffer pool 的事务和操作 都会 等待。 这个规则的例外情况是， 当缓冲池碎片整理时，允许对 缓冲池进行并发访问，而当 缓冲池 大小减小时，页面将被撤回(withdrawn)。 允许并发访问的一个缺点是，当页面被撤回时，它可能导致 可用页面的 暂时短缺

Nested transactions could fail if initiated after the buffer pool resizing operation begins.

Monitoring Online Buffer Pool Resizing Progress

```
mysql> SHOW STATUS WHERE Variable_name='InnoDB_buffer_pool_resize_status';
+----------------------------------+----------------------------------+
| Variable_name                    | Value                            |
+----------------------------------+----------------------------------+
| Innodb_buffer_pool_resize_status | Resizing also other hash tables. |
+----------------------------------+----------------------------------+
```

从 8.0.31 开始， 你也可以 监控 online buffer pool resizing 操作 通过 Innodb_buffer_pool_resize_status_code 和 Innodb_buffer_pool_resize_status_progress 状态变量。

Innodb_buffer_pool_resize_status_code 报告了 online buffer pool resizing 操作 的状态

0	没有resize 操作在执行
1	开始resize
2	禁用AHI中 (adaptive hash index)
3	撤回块 withdrawing blocks
4	要求 全局 lock
5	resizing pool
6	resizing hash

7	resizing failed

Innodb_buffer_pool_resize_status_progress 状态变量 表达了 每个阶段的 处理百分比。在 每个 buffer pool instance 被处理后，百分比 被更新。 阶段改变时，reset 为0。

下面的查询 返回 一个 string 值 表示 buffer pool resizing 过程， code 代表当前 stage，当前百分比：

```
SELECT variable_name, variable_value
 FROM performance_schema.global_status
 WHERE LOWER(variable_name) LIKE "innodb_buffer_pool_resize%";
```

buffer pool resizing 过程 也可以在 server error log 中看到

```
[Note] InnoDB: Resizing buffer pool from 134217728 to 4294967296. (unit=
134217728)
[Note] InnoDB: disabled adaptive hash index.
[Note] InnoDB: buffer pool 0 : 31 chunks (253952 blocks) was added.
[Note] InnoDB: buffer pool 0 : hash tables were resized.
[Note] InnoDB: Resized hash tables at lock_sys, adaptive hash index,
dictionary.
[Note] InnoDB: completed to resize buffer pool from 134217728 to 4294967296.
[Note] InnoDB: re-enabled adaptive hash index.
```

从8.0.31开始， 启动server 时，带上 --log-error-verbosity=3，打印 额外信息到 error log中。

Online Buffer Pool Resizing Internals
resizing 操作是 后台线程执行的。
当增加 buffer pool size时，resizing 操作：

- 增加 page 到 chunks (chunk size 通过 innodb_buffer_pool_chunk_size 定义)
- 转换 hash table, list, pointer 为 使用内存的新地址。
- 增加 新page 到 free list。

以上操作执行时，其他 访问 buffer pool 的线程 被阻塞。

当 减少 buffer pool size 时：

- 整理 buffer pool，撤销(释放) page
- 移除 chunk 中的 page (chunk size 通过 innodb_buffer_pool_chunk_size 定义)
- 转换 hash table，list，pointer 为 使用 内存的新地址。

在这些步骤中，只有 整理 buffer 和 撤销 page， 允许 其他线程 并发访问 buffer pool

15.8.3.2 Configuring Multiple Buffer Pool Instances
https://dev.mysql.com/doc/refman/8.0/en/innodb-multiple-buffer-pools.html

对于有 数个g 的 buffer pool，切分 buffer poll 为 多个 独立 实例 可以提示 并发性。

通过 innodb_buffer_pool_instances 来配置 多个 buffer pool 实例， 你可能也需要 调整 innodb_buffer_pool_size 的值。

当 innodb buffer pool 足够大，很多 数据请求 都可以 直接 从 内存中 检索。 你可能 遇到 瓶颈： 多个线程 同时访问 一个buffer pool。你可以 通过 使用多个buffer pool 来 减少这种竞争。
每个 page 通过 hash 随机选择 一个 buffer pool，然后保存进去。
每个 buffer pool 管理 它自己的 free list， flush list, LRU，以及 和buffer pool有关联的 所有 数据结构。
8.0之前，每个 buffer pool 通过它自己的 buffer pool mutex 进行保护。
8.0及之后， buffer pool mutex 被替换为 数个list 和 hash protecting mutex， 来降低竞争。

innodb_buffer_pool_instances 的值 在1 和64 之间(包含1,64，默认1)。 这个选项 只有当你 将 innodb_buffer_pool_size 设置为 >= 1G 时，才生效。你指定的是 总大小，会自动切分给 每个 buffer pool。
为了最好的性能， 指定 innodb_buffer_pool_instances 和 innodb_buffer_pool_size 后，每个 buffer pool instance 最好至少 1g。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-midpoint_insertion.html

15.8.3.3 Making the Buffer Pool Scan Resistant

比起使用严格的 LRU 算法， InnoDB 使用 一种技术 来 最小化 被加载到buffer pool中 而且不会再被访问 的数据。 目标是 确保： 经常被访问的 hot page 保留在 buffer pool中，即使 预读 和 全表扫描 会带来新的 数据块，这些数据块 以后 可能会 也可能不会 被访问。

新读取的 block 被 insert 到 LRU list 的 中部。
所有新读取的 page 被插入到 LRU list 中，默认 是 从尾部开始的 3/8 处。
当 buffer pool 第一次访问 page 时， page 被 移动到 list 的 头 (即最常使用的那端)。
LRU的 驱逐 和 标准的LRU一样。
这个设置 将 LRU list 分为 2段，插入点之下的 page 被认为是 old，是 LRU驱逐的理想目标

buffer pool 的内部工作，LRU 的详细说明， 可以看 15.5.1 buffer pool

你可以控制 LRU list 的 insert point， 可以选择 是否对 通过表扫描 或 通过index扫描而 读取到 buffer pool 中的 block 应用 相同的 优化。
innodb_old_blocks_pct 控制 LRU 中 old 快的 百分比。 默认是 37，表示 固定比例 3/8。(。。37%) 。取值范围是 5(新page会被很快驱逐出LRU) - 95(只有5%的空间留给了 hot page，使得 和 标准LRU 非常 近似)。

避免buffer pool 被提前读取 扰乱 的 优化 可以同样避免 由表或index 扫描 引起的 类似问题。
在这些扫描中，一个 data page 通常会被 快速访问几次，然后 再也不会被访问。
innodb_old_blocks_time 指定了 时间窗口(毫秒)，在首次访问这个 page 之后 的这段 时间窗口内 的访问，不会 导致 page 被移到 LRU头部。 默认是 1000， 增加这个值 会 使得 更

多的 block 更快地 老化。

innodb_old_blocks_pct 和 innodb_old_blocks_time 可以 在 mysql option文件 (my.cnf 或 my.ini) 中指定， 或 运行时 通过 set global 修改(这个需要足够的权限)。

SHOW ENGINE INNODB STATUS 展现 buffer pool的统计信息 来 归纳 这些参数的效果。更多信息，看 https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html#innodb-buffer-pool-monitoring

由于这些参数的 效果 很大程度上 依赖于 硬件，数据，负载，所以 在生产上 更改这些配置之前，一定要做基准测试 验证 有效性。

在混合工作负载中，大多数都是 OLTP 类型的，并且定期进行批处理 报告查询，这会导致 大量的扫描，在 批处理运行期间 设置 innodb_old_blocks_time 的值 有助于 将 正常工作的工作集 保留在 buffer pool中。
。。运行前设置下， 允许后恢复。。

当 扫描 不能完全容纳于 buffe pool 中的 大表时，设置 innodb_old_blocks_pct 到一个 小值 来让 这些 只会读取一次的数据 不太占用 buffer pool 的 地方。例如设置为5%

当扫描适合内存的小表时，把 page 放到 buffer pool 中 可以减少开销， 所以你可以 使用默认值，或 更高一点，比如 50%。

innodb_old_blocks_time 参数 的效果 比 innodb_old_blocks_pct参数 更难 预测，相对较小，并且随着工作负载的变化 而变化更大。
要获得最优值，进行你自己的基准测试，如果 调整 innodb_old_blocks_pct 的性能改进不够。

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-read_ahead.html
15.8.3.4 Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)
配置buffer pool 的预读

预读请求是一种IO 请求，用于 异步读取 buffer pool 中的多个 page，预测 后续会对这些 page 进行读取。 request 将所有page 放到 一个 extent。 InnoDB 使用 2个 预读算法 来提升 IO性能。
extend定义：A group of pages within a tablespace. For the default page size of 16KB, an extent contains 64 pages

。。。buffer pool 不是内存吗？ 从buffer pool 中读取多个 page ，放到哪里？ 不还是内存中吗？
A read-ahead request is an I/O request to prefetch multiple pages in the buffer pool asynchronously,
难道说，上面的是 异步读取page 到 buffer pool ？主要是 in，不是 into。 我感觉是 page (that) in the buffer pool 。 对是 (that)，下面也用到了 page in the buffer

Linear

linear预读 是一个技术，它预测 是按 顺序访问的，所以当前 访问的 buffer pool中的 page 的 后续page 很快会被访问。
你可以控制 InnoDB 在顺序读取多少个page后 触发 预读，by innodb_read_ahead_threshold。以前，没有这个参数的时候，InnoDB 只有 在 读取到 当前 extent 的 最后一个page 时，判断 是否 要 预读 后续的一整个 extent。
innodb_read_ahead_threshold 取值范围 [0,64]，默认56.
例如，如果你设置 48， 当 当前extent的48个page 被顺序 access 时，innodb 会触发一个 linear 预读请求。 如果设置为8，只需要extent中 8个page 被顺序访问，就会触发 linear 预读请求。
你可以在 mysql 配置文件中，或 set global 动态 调整 这个参数。

Random
随机预读，它 根据 buffer pool 中已有的 page 预测 几时这些page会被读取， 不关心 page 读取的顺序。
如果在 buffer pool 的 同一个extent 中发现 13个连续的page，InnoDB 会异步触发 一个 request 来 预加载 extent 中剩余的page。(。。不知道这个剩余的page 是指 13个中剩余的，还是 extent中所有其他的，估计后者。)。
通过 innodb_random_read_ahead 设置为 ON 来 启用这个 功能。

SHOW ENGINE INNODB STATUS 命令展示了 统计信息，帮你 评估 预读算法的 有效性。
统计信息 包含了 下列 全局状态变量的 counter 信息：
    Innodb_buffer_pool_read_ahead
    Innodb_buffer_pool_read_ahead_evicted
    Innodb_buffer_pool_read_ahead_rnd

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-flushing.html
15.8.3.5 Configuring Buffer Pool Flushing

InnoDB 在后台 执行一些task，包括 flush buffer pool中的dirty page。dirty page 是指那些 已经被修改 但是没有 写回磁盘的 page。

8.0中，buffer pool flushing 由 page cleaner 线程 执行。 page cleaner线程数量 由 innodb_page_cleaners 控制，默认4。 但是，如果 线程数量 超过 buffer pool 实例数量，innodb_page_cleaners 被自动调整为 等于 innodb_buffer_pool_instances。

当 dirty page 的百分比 达到 innodb_max_dirty_pages_pct_lwm 定义的 low water mark(低水位线)值时， 触发一次 buffer pool flushing。 默认是 10%， 你可以设置为 0，来禁用这个 early flushing 行为

innodb_max_dirty_pages_pct_lwm 的目的是，控制 buffer pool 中 dirty page 的百分比，防止 dirty page 的数量 超过 innodb_max_dirty_page_pct 定义的值，默认90。 如果 buffer pool 中 dirty page 的百分比 达到 innodb_max_dirty_page_pct 定义的值，innodb 会主动 flush buffer pool page。
。。？？？ 一个10%， 一个90%，感觉差好多。 感觉 90% 不可能达到的。

innodb_max_dirty_page_pct_lwm 的值应该 始终 小于 innodb_max_dirty_page_pct。

其他 可以 微调 buffer pool flushing 行为的 参数：

innodb_flush_neighbors，flush buffer pool 中的page 时，是否也一并 flush 同一个 extent 中的 其他 dirty page。

默认0，禁用。 在SSD 上推荐 0，因为 seek time(寻道时间) 很短。

1，flush 连续的 dirty page， 在同一个 extent 中的。

2，flush 同一个 extent 中的 dirty page。

当 table data 被存储在 HDD 上时，和 多次操作，每次flush一个page 相比， 在一次操作中 flusing neighor page 降低IO开销(主要是 磁盘寻道的 开销)。

在SSD中，seek time不是 关键因素，所以可以禁用。

。。？？？ pct,pct_lwm， 都是比例， 到达这个比例后，触发 flush，难道这个 flush 不会 flush 完？ 还是说这里的逻辑是：

```
for (dirty page : ALL dirty page in this extent)
{
    flush dirty page
    if (innodb_flush_neighbors == true)
    {
        if (neighbor is dirty)
        {
            recursive ?
        }
    }
}
```

。。就是 任何时候，flush 必然 flush完全部的 dirty page，只不过 加上这个配置后，会看下 旁边的是不是dirty?

innodb_lru_scan_depth，对于每个 buffer pool实例，page cleaner线程 在 buffer pool LRU list 上 扫描多深 来搜索 dirty page。 这是一个 后台操作， page cleaner 线程 每秒执行一次。

小于默认值的 设定值 通常适合大多数工作负载。 明显大于 必要值 的 设置定 会 影响性能。 如果在 通常的工作负载下，还有 空闲的IO，那么可以增加这个值。 如果 写入密集型，IO饱和了，那么减少值， 特别是在 大 buffer pool 的情况下。

当调整 innodb_lru_scan_depth， 从 低值开始，慢慢增长，目标是 很少看到 zeor free page。 在 修改 buffer pool 实例数时 也考虑调整 innodb_lur_scan_depth， 因为 innodb_lru_scan_depth * innodb_buffer_pool_instances 定义了 每秒执行的 page cleaner 线程的 所做工作的总量。

innodb_flush_neighbors 和 innodb_lru_scan_depth 主要是为了 写入密集型。

对于大量 DML 活动，flush 可能会落后，如果 不积极flush的话， 如果太积极，flush 可能占用 磁盘IO。

理想的值 取决于 你的 工作负载，数据访问模式，存储配置(HHD/SSD)

Adaptive Flushing

innodb 使用 自适应flushing 算法 来 基于redo log生成速度 和 当前flushing的 rate 动态调整 flushing 的 rate。 目的是为了 确保 flushing 和当前工作负载 保持同步，从而 使性能总体平稳。

自动调整flush频率，有助于避免 在buffer pool flush 导致的 IO 突发影响 普通读取活动可

用的IO 时 吞吐量 突然下降。
Sharp checkpoints,
。。。。。这章好多。。随便写了。
当innodb 想要重用 log 文件的 一部分时，会发生 sharp checkpoint。在 重用 部分log文件之前，这部分 的 redo log 的 dirty page 必须 flush。 如果 log文件变满，sharp checkpoint 发生，导致 吞吐量的下降。 即使 innodb_max_dirty_page_pct 没有达到，这个场景也可能发生。
自适应flush 可以避免这种情况，通过 跟踪 buffer pool 中 dirty page 的数量，及 redo log 记录被生成 的 rate。 基于这些信息， 决定 每秒 flush 多少个 dirty page。
。。就是说，不是 全部 flush完， 而是 flush 多少个后，就结束。。
innodb_adaptive_flushing_lwm，定义了 redo log 容量的 低水位。超过这个值后，adaptive flushing 被启动，即使 innodb_adaptive_flushing 是 禁用的。
innodb_adaptive_flushing，默认 启用
innodb_flushing_avg_loops，对于之前计算的 flushing 状态的 snapshot 可以保存 多少次迭代。 值大，说明之前计算的 snapshot 存活长，所以 adaptive flushing 不敏感，反应慢。
innodb_log_file_size
innodb_io_capacity
innodb_io_capacity_max

Limiting Buffer Flushing During Idle Periods
8.0.18开始，可以使用 innodb_idle_flush_pct 来 限制 在idle期间(这期间，database page 没有被修改)， buffer pool flushing 的rate。
可以延长硬件的生命

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-preload-buffer-pool.html
15.8.3.6 Saving and Restoring the Buffer Pool State
为了降低 重启服务器后的 预热时间， innodb 将 每个buffer pool 的 最近使用的page 的一定比例， 在 shutdown时 保存下来，在 startup 时 恢复这些page， 最近使用的page 的百分比 被 innodb_buffer_pool_dump_pct 配置。

。。跳

---
https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool-in-core-file.html
15.8.3.7 Excluding Buffer Pool Pages from Core Files

core file 记录了 一个运行中process 的 状态和 内存image。 由于 buffer pool 驻留在 主内存中，且 运行中process 的 内存image 会 dump 到 core file， 有着 大buffer pool 的 系统 会 生成 大 core file， 当 mysqld process die。

大 core file 是有问题的，因为 许多原因，包括：写入磁盘 消耗的时间，消耗的磁盘空间，传输大文件的挑战。

要降低 core file，你可以 禁用 innodb_buffer_pool_in_core_file 来 从 core dump 中 忽

略 buffer pool page。 这个变量 从 8.0.14 开始， 默认启用。

排除 buffer pool page 也符合安全性， 有时，你会 share core file 到 外部， for debugging 的目的。

禁用 innodb_buffer_pool_in_core_file 有效，只有当 core_file 启用，且 OS 支持 MADV_DONTDUMP non-POSIX 扩展 的 madvise() 系统调用，这个 从 Linux 3.4 开始支持。MADV_DONTDUMP 扩展 使得 core dumpe 中 指定range 的 page 被移除。

假设OS 支持 MADV_DONTDUMP，那么 下面的命令，生成的 core file 不包含 buffer pool page ：

```
$> mysqld --core-file --innodb-buffer-pool-in-core-file=OFF
```

core_file 是只读的，且 默认禁用。
innodb_buffer_pool_in_core_file 动态的， 可以在 启动时，或 运行是通过 set 来设置

```
mysql> SET GLOBAL innodb_buffer_pool_in_core_file=OFF;
```

。。跳，这里内容没什么用

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-thread_concurrency.html
15.8.4 Configuring Thread Concurrency for InnoDB

InnoDB 使用 OS 线程 来处理 来自用户事务的请求。(事务可能触发很多请求到InnoDB，在它们commit或rollback之前)。在 现代OS 和 多核服务器上， 上下文切换是 高效的， 大部分工作负载 在 不限制并发线程 的情况下是 工作良好的。

在有助于 最小化 线程间上下文切换 的情况下，InnoDB 使用 许多 技术 来 限制 并发执行的OS线程的 数量 (即，同时处理的request数量)。
当 InnoDB 从 用户session 收到 一个新request，如果 当前并发执行的线程 数量 达到 预先设置的限制， 新的request 会 sleep 短暂时间 然后 再次尝试。 sleep后还是无法 安排 的request 会 放入一个 FIFO 的queue，最终会被处理。 等待lock 的线程不会被 计入 并发执行中的线程的数量。
。。。有点儿像 非公平锁。

通过 innodb_thread_concurrency 来 限制 并发线程数量。 一旦 执行中的线程 数量 达到这个 限制， 额外的线程 会 sleep innodb_thread_sleep_delay 定义的 毫秒数， 然后 retry，然后 可能进入 queue。

你可以设置 innodb_adaptive_max_sleep_delay ，它是 你允许 innodb_thread_sleep_delay 的 最大值， innodb 会 根据当前线程调度活动 动态调整 innodb_thread_sleep_delay。

innodb_concurrency_tickets

innodb_adaptive_hash_index

。。跳

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-multiple_io_threads.html
15.8.5 Configuring the Number of Background InnoDB I/O Threads

innodb 使用 后台线程 来服务各种类型的 IO 请求。

innodb_read_io_threads
innodb_write_io_threads

这2个参数，默认4，取值范围是[1, 64]
只能在 启动时 通过 my.cnf 或 my.ini 设置，不能动态调整。

每个后台线程 可以处理 最多 256个 pending IO请求。
后台IO 的主要来源 是 预读request。
innodb 尝试平衡 incoming request 的负载 使得 大多数后台线程 相同地工作。
innodb 也尝试 将 从同一个extent中 读取的request 分配到 同一个 线程， 来增加 合并 request 的几率。

如果你有一个 high end IO subsystem，你会看到 超过 64 * innodb_read_io_threads 的 pengding read request， 在 show engine innodb status 的输出中， 你可以 增加 innodb_read_io_threads 的值 来 提升性能。

Linux 系统上，innodb 默认 使用 异步IO subsystem 来执行 对data file page 的 read-ahead 和 write 请求， 这改变了 innodb 后台线程 服务 这些类型的IO请求的方式。 更多信息看 15.8.6(下一章)

更多 innodb io 性能的 信息，看 8.5.8： optimizing innodb disk IO

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-linux-native-aio.html

15.8.6 Using Asynchronous I/O on Linux

innodb_use_native_aio

---

https://dev.mysql.com/doc/refman/8.0/en/innodb-configuring-io-capacity.html

15.8.7 Configuring InnoDB I/O Capacity

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-row-format.html

15.10 InnoDB Row Formats

表的 row format 决定了 它的row怎么 物理存储，影响了 query 和 DML 操作 的性能。
由于 单个 disk page 容纳 更多的row，query 和 index lookup 能更快，buffer pool中的 cache memory 需要得更少， 写入更新后值的 IO请求更少。

每个表中的 data 被 分为 page。 构成每个table 的page 以 B-tree index 的 数据结构 安排。
表数据 和 二级index 也都使用这种类型的 结构。
The data in each table is divided into pages. The pages that make up each table are arranged in a tree data structure called a B-tree index. Table data and secondary indexes both use this type of structure.
。。很多都说是 B+树啊。 但是这里说 B 树。

代表 整个表 的 B树index 被称为 聚集index(clustered index)，它根据 主键 进行组织。
聚集index 数据结构 的 node 包含了 这row 的所有column的值。
二级index 数据结构 的 node 包含了 index列的值 和 主键列。
The nodes of a clustered index data structure contain the values of all columns in the row.
The nodes of a secondary index structure contain the values of index columns and primary key columns.

。。B+树 只有叶子节点保存值， B树 所有节点都保存值。

variable-length column(可变长度列) 是 列值都保存在B树index节点 这个规则的例外。
由于太长 而无法 放入 B树 page 的 可变长度列 被保存在 称为 overflow page 的 单独分配的磁盘 page中。 这些column 被称为 off-page column (页外列)。
off-page column 的值 被存储在 overflow page 组成额 单链表中，每个这样的列 都有 它自己的 由一个或多个 overflow page 组成的 list。
根据 column 长度， variable-length column 值的 全部 或 prefix 被存储到 B树 来避免浪费 存储 和 不得不多读一个/多个 page。

innodb存储引擎 支持 4种 row format: redundant 冗余, compact 紧密, dynamic 动态, compressed 压缩

Table 15.15 InnoDB Row Format Overview

Row Format	Compact Storage Characteristics 紧凑的存储特性	Enhanced Variable-Length Column Storage 增强可变长列的存储	Large Index Key Prefix Support 大索引key前缀的支持	Compression Support 压缩	Supported Tablespace Types
REDUNDANT	No	No	No	No	system, file-per-table, general
COMPACT	Yes	No	No	No	system, file-per-table, general
DYNAMIC	Yes	Yes	Yes	No	system, file-per-

					table, general
COMPRESSED	Yes	Yes	Yes	Yes	file-per-table, general

REDUNDANT Row Format
redundant 格式 提供了 对早期 mysql 的兼容。

使用redundant row format 的 table， 对于 可变长列 (varchar,varbinary,blob,text)的值 值 存储 前768 个byte 到 B树节点的 index 记录中， 剩余的 保存到 overflow page。
长度 >= 768 字节 的 fixed-length 列， 被编码为 可变长列，可以在 off-page 中存储。例如，一个 char(255) 列，可以超过 768 byte，如果 一个char 超过 3个byte，就像 utf8mb4 。。为什么不是 >768 的固定长度列 被 编码为 可变长列。 768 并不会 导致 overflow page 啊。

如果列的值 <= 768 byte，不会使用 overflow page，这样 所有的值都存储在B树 node中，可以节约 IO。 这 对于相对较小的BLOB 列值 很有效， 但是 可能导致 B树的节点中 充满了 data 而不是 key value(键值)， 降低 效率。
有许多 BLOB列的 表 会导致 B树节点 变得 太满，包含了 太少的node，使得 整个index 比 row were shorter 或 列值存储在off-page 中的 更 低效。

REDUNDANT Row Format Storage Characteristics
redundant 有下列存储特性

1. 每个index record 包含一个 6byte 的header。用来 将 连续record link到一起，也用于 row-level locking。
2. 聚集index 中的 record 包含 所有 用户定义column 的 field。额外还有 6 byte 的 事务ID field，和 7byte 的 roll pointer field
3. 如果 table 没有定义 主键，每个 聚集index record 也包含 6byte 的 row ID field。
4. 每个二级index record 包含 为 不在二级index 中的 聚集index key 定义的 所有 主键列。Each secondary index record contains all the primary key columns defined for the clustered index key that are not in the secondary index.
5. record 包含了 对record 的每个 field 的 指针。如果 record 中的 field 总长度 < 128 byte，指针是 一个byte长度，否则，2个byte。 pointer的数组 被称为 record directory。 指针指向的 区域 是 record 的数据部分。
6. 在内部，固定长度字符列 如char(10) 以 固定长度格式 存储。varchar列的 后缀空格不会被移除。
7. 大于等于 768 字节的 固定长度列 被编码为 变长列，可以被存储到 off-page。
8. SQL 的 NULL 值 在 record directory 中 占据 1 或2 byte。如果存储在 变长列中，SQL NULL值 占用 record的data part 的 0 个byte。 对于 固定长度列，在 record 的 data part 中 占据 固定长度。为NULL 值 占据 固定长度空间 可以 允许 列 可以原地 从 NULL 变成 non-NULL 值，不会 导致 index page fragmentation(分裂)。

COMPACT Row Format
对比 redundant， compact 可以降低 20%的 存储空间，但是 某些操作 需要更多的CPU算力。
如果你的 工作负载是 典型的那种： 被cache命中率 和 磁盘速度 限制的， compact 会更快。 如果 工作负载 被 CPU限制了， 那么 compact 更慢。

使用 compact 的表，保存 变长列(varchar,varbinary,blob,text)的 前768个byte 到 B树节点的 index record 中， 剩余的 保存到 overflow page。
>= 768字节的 固定长度列 被编码为 变长列，可以保存到 off-page中。例如，char(255)可以超过768个byte，如果 一个char 占用超过 3个byte，utf8mb4编码。
。。和 redundant 一样。
如果列的值 <= 768 byte，不会使用 overflow page，这样 所有的值都存储在B树 node中，可以节约 IO。 这 对于相对较小的BLOB 列值 很有效， 但是 可能导致 B树的节点中 充满了 data 而不是 key value(键值)， 降低 效率。
有许多 BLOB列的 表 会导致 B树节点 变得 太满，包含了 太少的node，使得 整个index 比 row were shorter 或 列值存储在off-page 中的 更 低效。
。。直接复制redundant。

COMPACT Row Format Storage Characteristics

1. 每个 index record 包含 5 byte header，它的前面可能有一个 变长header。header用来 link 连续的record，也用来 row-level linking
2. record header 的 变长 部分 包含 bit vector 用来 指示 NULL 列。如果 index 中的 可以为NULL 的 column 的数量是 N，那么 bit vector 占据 CEILING(N/8) byte。 NULL 列 除了这个 vector中的 bit 外 不会再占据 任何空间。 header的变长部分 也包含了 变长列 的长度。 每个长度 占据 1或2 byte，依赖于 column 的最大长度。 如果 index 中所有列 都是 NOT NULL 且 都有 固定长度， 那么record header 没有 变长部分。
3. 对于每个 non-NULL(。。这个应该是 实际非NULL，而不是定义的) 变长 field，record header 以 1或2 byte 的空间 存储 列的 长度。只有 当 column 的一部分 存储到 overflow page 中 或 最大长度超过255byte 且实际长度超过127 byte时 才需要 2byte。对于 外部存储的 column， 2 byte 长度 表示 内部存储的part 加上 指向外部存储part 的 20byte 的指针 的 长度。内部part 是768 byte，所以 长度是 768 + 20。 20 byte 指针 存储了 列的真实长度。
4. record header 之后 是 non-NULL 列的 data
5. 聚集index 中的 record 包含了 所有 用户定义列的 field。以及 额外的 6 byte 的事务ID， 7 byte 的 roll pointer
6. 如果 没有定义 主键，每个 聚集index record 依然 包含 6 byte 的 row ID
7. 每个 二级index record 包含了 所有 为 聚集index key 定义的 且 不在 二级index 中的 主键列。 如果 主键列中 有 变长列，每个 二级index 的 record header 有一个 变长部分 来 记录 它们的长度，即使 二级index 是定义在 定长列上的。
8. 在内部，对于 非变长 字符集， 定长字符列，如 char(10) 被存储在 定长格式。varchar 列中的 后缀空格不会被移除。
9. 在内部，对于 变长 字符集，如utf8mb3 和 utf8mb4，innodb尝试 通过移除尾部空格 来保存 char(N) 到 N个byte。 如果 char(N) 列的 byte长度 超过 N byte，将 尾部空格移除 为 列值的byte 长度的最小值。char(N) 列的 最大长度 为 最大的字符byte长度 * N。
   为 char(N) 保留 最小的N byte。 在许多情况下，保留最小的 N byte 已经能够 使得 更新 在原地发生，不会导致 index page fragmentation(分裂)。 相比之下，在使用 redundant时， char(N) 列 需要占据 最大char字节长度 * N 的空间。
   >=768字节的 固长列 被编码为 变长field，可以被储存到 off-page。

DYNAMIC Row Format

dynamic 提供了 和 compact 一样的 存储特性，但增加了 对 long变长列 的 存储能力的 增强， 和 large index key prefix 的 支持。

当 使用 ROW_FORMAT=DYNAMIC 创建表时， innodb 可以 将 long变长列的值 全部 off-page 存储，同时 在 聚集index record 中 包含 一个 只有20 byte 的指针 来指向 overflow page。 >= 768 byte 的 固长field 被编码为 变长field。

column 是否 off-page 存储，取决于 page size 和 row 的 total size。 当 row 太长时，选择 最长的column 进行 off-page 存储 直到 聚集index record 能放入 B树 page。
<= 40 byte 的text 和 blob 列也存储在 line(行) 中

dynamic row format 保持了 存储整row 到 index node (如果可以放入的话) 的 高效 (就像 compact 和 redundant 一样)， 但 dynamic 避免了 long column 的 数据byte过大 而 无法放入 B树节点 的问题。dynamic 基于这种思想：如果 长数据值的 一部分 需要 存放在 页外，那么通常，将 这个值完全 存储到 off-page 是 最有效的。
通过 dynamic format， 更短的column 更可能 保留在 B树节点中，最小化 row 所需的 overflow page 所需的 数量。

dynamic 支持 index key prefix 最多 3072 byte。

使用 dynamic row format 的 表 能被存储 到 system 表空间，file-per-table 表空间，general 表空间。
要存储 dynamic table 到 system表空间， 要么 禁用innodb_file_per_table和普通的create table 或 alter table 语句， 要么 在 create table 或 alter table 中 使用 tablespace [=] innodb_system 这个 table option。
innodb_file_per_table 不适用于 general 表空间，也 不适用于 通过tablespace [=] innodb_system 在 system 表空间存储的 dynamic 表。

DYNAMIC Row Format Storage Characteristics
dynamic 是 compact 的一个变种， 对于 存储特征，请查看 compact 的

COMPRESSED Row Format
compressed 提供了 和 dynamic 相同的 存储特征和能力， 但是 增加了 对 表和index数据压缩的 支持。

compressed 的 off-page存储的 内部细节 和 dynamic 类似，额外考虑了 table 和 index 数据压缩 来使用 更小的 page size。
KEY_BLOCK_SIZE 控制 column 数据 如何存储到 聚集index 中，在 overflow page 中 占据多少空间。 更多信息：15.9 innodb table and page compression

compressed 支持 最多 3072 byte 的 index key prefix。

使用 compressed row format 的 table 可以存储到 file-per-table 表空间 或 general 表空间。 system 表空间 不支持 compressed。
要存储到 file-per-table 表空间， innodb_file_per_table 必须被 启用。
general 表空间 支持所有row format， 但注意： 由于 物理page size的不同， 压缩和未压缩的 表 不能共存于 同一个 general tablespace。

Compressed Row Format Storage Characteristics
compressed 是 compact 的一个变种。 看 compact row format storage characteristics

Defining the Row Format of a Table
innodb 的 默认 row format 通过 innodb_default_row_format 定义，默认是 dynamic。
当 row_format 没有被定义 或 row_format=default 时， 使用 默认的 row format。

可以在 create table 或 alter table 语句中 显式 使用 row_format 来指定 row format。

```
CREATE TABLE t1 (c1 INT) ROW_FORMAT=DYNAMIC;
```

innodb_default_row_format 可以动态set

```
mysql> SET GLOBAL innodb_default_row_format=DYNAMIC;
```

如果没有 指定 row_format，或 row_format=default， rebuild table 的操作 会修改 表的 row format 为 innodb_default_row_format。
table-rebuilding 操作 包括 使用algorightm=copy 或 algorithm=inplace 的 alter table。optimize table 也是 table rebuild operation.

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME LIKE 'test/t1' \G
```

在将 现有的 redundant 或 compact 表 转为 dynamic 之前，考虑 下面的潜在的 问题

1. redundant 和 compact 支持 最多 767 byte 的 index key prefix， 而 dynamic 和 compressed 支持 3072 byte 的 index key prefix。 在复制架构中， 如果 source 是 dynamic，副本是 compact， 下面的 DDL 没有显式定义 row format，在source 可以成功，但是 replica 会失败。
   ```
   CREATE TABLE t1 (c1 INT PRIMARY KEY, c2 VARCHAR(5000), KEY i1(c2(3070)));
   ```
   更多信息，看15.22 innodb limites
2. import 一张没有显式定义 row format 的表， 会导致 schema mismatch error ，如果 source server 和 target server 的 innodb_default_row_format 不同。

Determining the Row Format of a Table
通过 show table status 确定 表的 row format

```
mysql> SHOW TABLE STATUS IN test1\G
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Dynamic
           Rows: 0
 Avg_row_length: 0
    Data_length: 16384
Max_data_length: 0
   Index_length: 16384
      Data_free: 0
 Auto_increment: 1
    Create_time: 2016-09-14 16:29:38
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_0900_ai_ci
```

```
        Checksum: NULL
  Create_options:
         Comment:
```

查询 information schema 的 innodb_tables

```
mysql> SELECT NAME, ROW_FORMAT FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE
NAME='test1/t1';
+----------+------------+
| NAME     | ROW_FORMAT |
+----------+------------+
| test1/t1 | Dynamic    |
+----------+------------+
```

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-disk-management.html

15.11 InnoDB Disk I/O and File Space Management

- 15.11.1 InnoDB Disk I/O
- 15.11.2 File Space Management
- 15.11.3 InnoDB Checkpoints
- 15.11.4 Defragmenting a Table
- 15.11.5 Reclaiming Disk Space with TRUNCATE TABLE

as a dba，你必须管理 磁盘IO 来 放置 IO子系统 饱和， 管理 磁盘空间 以避免 存储器满。

ACID 的设计模型， 需要 一定量的 看起来 冗余的 IO，但是 这些IO 有助于 确保 数据 可靠性。

本节讨论 innodb 的 IO 和 磁盘空间。

- 控制 后台IO的数量 来提升 query 性能
- 启用 或 禁用： 消耗额外IO 提供 额外的durability 的 feature。
- 将 table 组织成 许多小文件，一些大文件，或者 2者的混合。
- 平衡 redo log 的 size 和 log文件文件变满时的IO活动。
- 如何重新组织 table 来获得 最优的 查询性能。

==========

跳过了
InnoDB and Online DDL， 介绍了所有 在线DDL操作，性能，并发，空间，内存管理，并发线程，简化，失败条件，限制

InnoDB Data-at-Rest Encryption，

InnoDB Startup Options and System Variables，包括 所有的 启动参数 及 详细介绍。 启动参数列表 复制到 OneNote 的 MySQL-option 中

==========

==========

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema.html
15.15 InnoDB INFORMATION_SCHEMA Tables

innodb 的INFORMATION_SCHEMA 的表 提供了 metadata，状态信息，innodb存储引擎各方面的统计信息。
可以通过 下面语句 查看 information_schema 的表

mysql> SHOW TABLES FROM INFORMATION_SCHEMA LIKE 'INNODB%';

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-compression-tables.html

15.15.1 InnoDB INFORMATION_SCHEMA Tables about Compression

有 2对 有关 压缩 的 innodb information_schema 表，可以深入了解 压缩的总体工作情况

innodb_cmp 和 innodb_cmp_reset，提供了 关于 压缩操作的数量和执行压缩消耗的时间。

innodb_cmpmem 和 innodb_cmpmem_reset，提供了 为压缩分配内存的方式，在buffer pool 申请的 压缩page 的信息。

。。具体介绍 跳

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-transactions.html

15.15.2 InnoDB INFORMATION_SCHEMA Transaction and Locking Information

15.15.2.1 Using InnoDB Transaction and Locking Information

15.15.2.2 InnoDB Lock and Lock-Wait Information

15.15.2.3 Persistence and Consistency of InnoDB Transaction and Locking Information

注意，本章描述的 用于展示locking 信息的 performance schema 的 data_locks 和 data_lock_waits 表， 从8.0 开始 就 information_schema 的 innodb_locks 和 innodb_lock_waits 取代了。

一个 information_schema 的表 和 2个 performance schema 的表 允许你 监控 innodb 事务 和 诊断潜在的locking 问题：

innodb_trx：属于 information_schema，提供了 innodb 中当前正在执行的 所有事务 的信息， 包括 事务状态(比如，执行中，等待lock)，事务正在执行的sql语句。

data_locks：这个 performance schema 表，为 每个 hold lock 和 每个 阻塞等待lock 释放的 lock request 生成一条row。

每个hold lock 一行，不管 hold 这个lock的 事务的 状态 (innodb_trx.trx_status 是 running, lock wait, rolling back, committing)

每个等待其他事务释放锁的 事务(innodb_trx.trx_status 是 lock wait) 被 一条 blocking lock request 阻塞。 blocking lock request 是 为了 以不相容模式 获得 其他事务 hold着的 row or table lock。 lock request 总是 有着 和 阻塞 request 的 被hold的lock 的模式 不兼容的模式。(read vs write, shared vs exclusive)

阻塞的事务 无法执行 直到 其他事务 commit 或 roll back 后释放 被请求的 lock。 对于每个 阻塞的 事务，data_locks 包含 一条row 描述了 事务请求的每个

lock，以及 它正在等待的 每个锁。

data_lock_waits：performance schema 表 表明 对于给定的lock，哪些事务正在waiting，或 一个给定的事务 在等待 什么lock。 表中 为 每个阻塞的事务 创建一条row，包含： 它已经申请的 lock 和 阻塞这个request的 任何lock。
requesting_engine_lock_id 指向了 事务 请求的 lock， blocking_engine_lock_id 指向了 其他事务hold的，阻止本事务继续执行的 lock。
对于任何给定的 阻塞的事务，data_lock_waits 中的 所有row 都有相同的requesting_engine_lock_id, 不同的 blocking_engine_lock_id。

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-examples.html
15.15.2.1 Using InnoDB Transaction and Locking Information

注意，本节，描述了 perfoamce schema 中的 data_locks 和 data_lock_waits 表，从8.0开始，变成了 information schema。

Identifying Blocking Transactions

```
SELECT
  r.trx_id waiting_trx_id,
  r.trx_mysql_thread_id waiting_thread,
  r.trx_query waiting_query,
  b.trx_id blocking_trx_id,
  b.trx_mysql_thread_id blocking_thread,
  b.trx_query blocking_query
FROM       performance_schema.data_lock_waits w
INNER JOIN information_schema.innodb_trx b
  ON b.trx_id = w.blocking_engine_transaction_id
INNER JOIN information_schema.innodb_trx r
  ON r.trx_id = w.requesting_engine_transaction_id;
```

或更简单：

```
SELECT
  waiting_trx_id,
  waiting_pid,
  waiting_query,
  blocking_trx_id,
  blocking_pid,
  blocking_query
FROM sys.innodb_lock_waits;
```

。。还有很多数据的分析。

Identifying a Blocking Query After the Issuing Session Becomes Idle
在识别 blocking 事务时，NULL 值代表 blocking query 所在的 session 已经 idle。此时，使用下列步骤 识别 blocking query

1. 识别 blocking 事务的 processlist id。在 sys.innodb_lock_waits 表中， 阻塞中的事务的 processlist ID 被展现在 blocking_pid 列中。
2. 使用 blocking_pid， 查询 mysql 的 performance schema 的 threads 表 来 确定 blocking 事务的 thread_id：

```
SELECT THREAD_ID FROM performance_schema.threads WHERE PROCESSLIST_ID = 6;
```

3. 使用 thread_id，查询 perfoamance schema 的 events_statement_current 表 来决定线程 执行的 最后一个 query：

```
SELECT THREAD_ID, SQL_TEXT FROM
performance_schema.events_statements_current
WHERE THREAD_ID = 28\G
```

4. 如果 线程执行的最后一个query 还不足以 确定 为什么 lock 被 hold，你可以 查询 performance schema 的 events_statements_history 表 来查询 线程 执行的 最后10个语句

```
SELECT THREAD_ID, SQL_TEXT FROM
performance_schema.events_statements_history
WHERE THREAD_ID = 28 ORDER BY EVENT_ID;
```

Correlating(关联) InnoDB Transactions with MySQL Sessions
。。挺烦的。

==========
https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-understanding-innodb-locking.html
15.15.2.2 InnoDB Lock and Lock-Wait Information

当事务 update 表中的一行，或 使用 select .. for update 对行加锁， innodb 会在那行上建立 lock 组成 的 list或queue。 类似地，对于表上的 table-level lock，innodb 也维护了 lock组成的 list。
如果第二个事务 想要 update row 或 以不兼容的模式 lock 已经被之前的事务 lock 的锁，innodb 增加一个 对row 的 lock request 到对应的 queue中。

For a lock to be acquired by a transaction, all incompatible lock requests previously entered into the lock queue for that row or table must be removed (which occurs when the transactions holding or requesting those locks either commit or roll back).
。。？

。。跳

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-internal-data.html

15.15.2.3 Persistence and Consistency of InnoDB Transaction and Locking Information

Data might not be consistent between the INNODB_TRX, data_locks, and data_lock_waits tables.

Data in the transaction and locking tables might not be consistent with data in the INFORMATION_SCHEMA PROCESSLIST table or Performance Schema threads table.

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-system-tables.html

15.15.3 InnoDB INFORMATION_SCHEMA Schema Object Tables

INNODB_DATAFILES
    innodb 的 file-per-table 和 general 表空间 的 data file path 信息
INNODB_TABLESTATS
    提供了 有关innodb 表的低级状态信息的视图，这些innodb表 是从 内存数据结构 派生的。
INNODB_FOREIGN
    定义在 innodb 表中的 外键的 metadata
INNODB_COLUMNS
    innodb table column 的 metadata
INNODB_INDEXES
    index 的metadata
INNODB_FIELDS
    innodb index 的 key column(field) 的 metadata
INNODB_TABLESPACES
    file-per-table,general,undo 表空间 的 metadata
INNODB_TABLESPACES_BRIEF
    innodb 表空间的 metadata 的子集
INNODB_FOREIGN_COLS
    定义在 innodb 表中的 外键的 column 的metadata
INNODB_TABLES
    关于 innodb 表的 metadata

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME='test/t1' \G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_COLUMNS where TABLE_ID = 71\G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_INDEXES WHERE TABLE_ID = 71 \G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_FIELDS where INDEX_ID = 112 \G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLESPACES WHERE SPACE = 57 \G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_DATAFILES WHERE SPACE = 57 \G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLESTATS where TABLE_ID = 71 \G
```

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_FOREIGN \G
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_FOREIGN_COLS WHERE ID = 'test/fk1'
\G

mysql> SELECT a.NAME, a.ROW_FORMAT,
        @page_size :=
         IF(a.ROW_FORMAT='Compressed',
          b.ZIP_PAGE_SIZE, b.PAGE_SIZE)
          AS page_size,
         ROUND((@page_size * c.CLUST_INDEX_SIZE)
          /(1024*1024)) AS pk_mb,
         ROUND((@page_size * c.OTHER_INDEX_SIZE)
          /(1024*1024)) AS secidx_mb
       FROM INFORMATION_SCHEMA.INNODB_TABLES a
       INNER JOIN INFORMATION_SCHEMA.INNODB_TABLESPACES b on a.NAME = b.NAME
       INNER JOIN INFORMATION_SCHEMA.INNODB_TABLESTATS c on b.NAME = c.NAME
       WHERE a.NAME LIKE 'employees/%'
       ORDER BY a.NAME DESC;
```

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-fulltext_index-tables.html

15.15.4 InnoDB INFORMATION_SCHEMA FULLTEXT Index Tables

下面的表提供了 fulltext index 的 metadata。

```
mysql> SHOW TABLES FROM INFORMATION_SCHEMA LIKE 'INNODB_FT%';
+-----------------------------------------+
| Tables_in_INFORMATION_SCHEMA (INNODB_FT%) |
+-----------------------------------------+
| INNODB_FT_CONFIG                        |
| INNODB_FT_BEING_DELETED                 |
| INNODB_FT_DELETED                       |
| INNODB_FT_DEFAULT_STOPWORD              |
| INNODB_FT_INDEX_TABLE                   |
| INNODB_FT_INDEX_CACHE                   |
+-----------------------------------------+
```

。。。

==========

15.15.5 InnoDB INFORMATION_SCHEMA Buffer Pool Tables

```
mysql> SHOW TABLES FROM INFORMATION_SCHEMA LIKE 'INNODB_BUFFER%';
+-------------------------------------------+
| Tables_in_INFORMATION_SCHEMA (INNODB_BUFFER%) |
```

```
+-------------------------------------------+
| INNODB_BUFFER_PAGE_LRU                    |
| INNODB_BUFFER_PAGE                        |
| INNODB_BUFFER_POOL_STATS                  |
+-------------------------------------------+
```

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-metrics-table.html

15.15.6 InnoDB INFORMATION_SCHEMA Metrics Table

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME="dml_inserts" \G
mysql> SELECT name, subsystem, status FROM INFORMATION_SCHEMA.INNODB_METRICS ORDER BY NAME;
```

==========

15.15.7 InnoDB INFORMATION_SCHEMA Temporary Table Info Table

==========

15.15.8 Retrieving InnoDB Tablespace Metadata from INFORMATION_SCHEMA.FILES

==========

==========

15.17 InnoDB Monitors

15.17.1 InnoDB Monitor Types

innodb 监视器有2种类型
标准 innodb 监视器，展示了下面的信息
- 主后台线程 完成的 工作
- semaphore waits
- 最近的 foreign key 和 死锁错误的 数据
- Lock waits for transactions
- 激活的事务 hold 的 表和record lock
- pengding IO操作 和 相关统计信息
- insert buffer 和 自适应hash index 统计信息
- redo log data
- buffer pool 统计信息
- row operation data

innodb lock 监视器，打印 额外的 lock 信息 作为 标准innodb 监视器输出的 一部分

monitor 的启用

monitor 的输出

==========
https://dev.mysql.com/doc/refman/8.0/en/innodb-backup.html
15.18.1 InnoDB Backup

热备份，冷备份，使用mysqldump的逻辑备份

15.18.2 InnoDB Recovery

Point-in-Time Recovery 。。恢复到现在。
要 将 innodb 数据库 从 物理备份制作的 时间 恢复到 现在， 你必须 以 启用 binary logging 的方式 运行 mysql server，即使在生成备份前。
在 恢复 备份后，要实现 point-in-time recovery， 你可以 应用 binary log 中 在备份制作后的 的 change。

Recovery from Data Corruption or Disk Failure，数据损坏或磁盘故障
如果是数据损坏，首先找到 一个没有损坏的备份。恢复这个备份，使用 mysqlbinlog 和 mysql 从 binary log file 做一个 point-in-time recovery 来恢复 备份制作时 到现在的修改。

一些情况下， 足够 dump,drop,re-create 一个或几个 损坏的table， 你可以使用 check table 来检查 table 是否损坏，但是 check table 不能检测所有的损坏。
一些情况下，可以 显而易见，数据库page损坏，是由于 OS的 file cache 损坏了，但是 disk 上的数据 是正确的。 最好先 重启电脑。这样 可能可以消除错误。 如果 mysql 由于 innodb 一致性问题 而无法启动，查看 15.21.3，forcing innodb recovery， 以 recovery 模式 启动 instance，它可以让你 dump 数据。

InnoDB Crash Recovery
要从 mysql server 的一次意外退出 中恢复，需要做的 仅仅是 重启 mysql server。
innodb 自动检查 log ，让 数据库 前滚 到 现在。
innodb 自动 回滚 崩溃时 未commit的事务。

innodb 崩溃恢复包括数个步骤：

1. tablespace discovery
   innodb用 tablespace discovery 来 识别 需要 redo log application 的 表空间。
2. redo log application
   redo log application 在初始化期间执行 (在接受任何 连接前)，如果 关机或崩溃时 所有的change 都已经 从 buffer pool 中 flush 到了 表空间(ibdata, ibd 文件)，redo log application 被跳过。 如果启动时 redo log 文件丢失，也跳过 redo log application
   每次 值改变时， 当前 最大自增counter的值 被写入到 redo log，这使得 它是 crash-safe。 在 恢复期间，innodb 扫描 redo log 来收集 counter value 的修改 ，并 应用这些修改 到 内存中的 表对象中。 更多关于 innodb 如何处理 自动自增value 的信息，看15.6.1.6 AUTO_INCREMENT handling in innodb。

   当遇到 index tree 损坏，innodb 将 损坏标记 写入到 redo log，这使得 损坏标记 crash-safe。 innodb 在每个 checkpoint上 也会 将 存储在内存中的 损坏标记 写入到 engine私有的系统表中。 在 恢复期间， innodb 从上述2个地方 读取 损坏标记，并在将 内存表 和 索引对象 标记位损坏 之前 合并结果。

   不推荐 通过移除redo log 来加速 recovery，即使 部分数据的损失 是可接受的。移除 redo log 应该 只有在 一次 innodb_fast_shutdown 为 0 或1 的 clean shutdown 后 考虑

3. 未完成数据的 回滚
   未完成事务 是 在意外的exist 或 fase shutdown 时 有效的事务。回滚未完成事务所需的时间 可能是 事务中断前 活动时间的 3-4倍，具体取决于 服务器负载。
   你不能 cancel 已经 回滚的事务。 在极端的情况下，当 回滚事务 被认为 需要 很长时间，那么 使用 innodb_force_recovery 为 3或更高 来启动 innodb 可能更快。

4. 合并change buffer
   应用 change buffer(系统表空间的一部分) 中的 change 到 二级index 的 leaf page，as index page 被读到 buffer pool 中

5. purge

删除 标记为 delete 的record，这些 record 对于 active 事务 已经不看见了。

在 redo log appliction 之后的步骤 不依赖 redo log ， 它们 以 正常处理的方式 并发 执行。只有 未完成事务的回滚 对于 crash recovery 是 特殊的。 insert buffer merge 和 purge 在 正常处理中 执行。

在 redo log application 之后， innodb 尝试 尽可能早 地 接受连接，来减少停机时间。
作为 crash recovery 的一部分，innodb 回滚那些 在server exit 时 没有 commit 或 处于 XA PREPARE状态的 事务。 回滚 由后台线程执行，和 新connection的 事务 并发执行。在 回滚完成前，新connection 可能遇到 locking conflict。

在绝大多数情况下，即使 mysql server 在 高负荷时 被 意外kill， recovery process 会自动发生，并不需要 DBA 做任何事。
如果 硬件错误 或 严重系统错误 损坏了 innodb 数据， mysql 可能拒绝启动，此时 15.21.3 forcing innodb recovery。

更多关于 binary log 和 innodb crash recovery，看 5.4.4 the binary log

Tablespace Discovery During Crash Recovery
如果，在recovery 期间，innodb 遇到 从上次checkpoint开始 写的 redo log，redo log 必须被应用到 受影响的 表空间。 受影响的表空间 的 识别 被称为 tablespace discovery。

tablespace discovery 依靠 innodb_directories 设置，它定义了 启动时 扫描 tablespace 文件的 目录。默认是 null，但是 innodb_data_home_dir，innodb_undo_directory，datadir 定义的 目录 一定会被 append 到 innodb_directories 中。
Recovery is terminated if any tablespace file referenced in a redo log has not been discovered previously.

==========

15.19 InnoDB and MySQL Replication
可以使得 副本的存储引擎 和 source的存储引擎 不同。

更多信息
17.4.4 using replication with different source and replica storage engines
17.1.2.6 setting up replicas
17.1.2.5 choosing a method for data snapshots

source上失败的 事务 不会影响 副本。
mysql 的复制 是基于 binary log，mysql将 修改数据的 sql 写入到 binary log中。事务的失败 不会写到 binary log， 所以不会 发送到 replica。
查看13.3.1 start transaction,commit,and rollback statements

Replication and CASCADE.
source 上的 innodb 表 的 行为 cascade 到 replica， 只有当 source 和 replica 使用了 innodb 的 外键关联。 无论是 基于语句 还是 基于row 的 replication 都是这样。

假设，你开始 复制，source 上创建 2个表，默认使用 innodb，使用下面的 create table 语句

```
CREATE TABLE fc1 (
    i INT PRIMARY KEY,
    j INT
);

CREATE TABLE fc2 (
    m INT PRIMARY KEY,
    n INT,
    FOREIGN KEY ni (n) REFERENCES fc1 (i)
        ON DELETE CASCADE
);
```

如果副本 的默认存储引擎 是 MyISAM， 创建相同的表，但是 使用了 MyISAM， foreign key 被忽视了。

。。。略，反正就是 source上 执行 delete from fc1 后， 通过外键cascade，将 fc2 的数据也删除了。replica 没有 外键，所以 fc2 的数据 没有被删除

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-memcached.html
15.20 InnoDB memcached Plugin

The InnoDB memcached plugin is deprecated as of MySQL 8.0.22; expect support for it to be removed in a future version of MySQL.

。。。看标题，还觉得很不错的功能，结果已经要移除了。。

==========

https://dev.mysql.com/doc/refman/8.0/en/innodb-troubleshooting.html

15.21 InnoDB Troubleshooting

==========

15.22 InnoDB Limits

一个表 最多1017列， virtual generated column 也包含在内。
最多 64个 二级index
dynamic，compressed row format 的 index key prefix length 最多 3072bytes
redundant，compact 的最多 767 bytes。
长度超过限制会报错
如果你 在创建mysql instance时 通过 innodb_page_size 降低 page size 到 8kb 或 4kb，index key 的最大长度 会按比例降低， 2072byte 是针对 16kb的， 所以8kb时是 1536 byte，4kb时是 768byte。
应用到 index key prefix 的限制 也应用到 full-column index key。

多列index 最多允许 16列。 超过则报错。

最大 row size (排除 保存在 off-page 的变长列)， 稍稍小于 page(4,8,16,32kb) 的 一半。例如，innodb_page_size 默认的 16kb 的 最大row size 是 8000 bytes。
但是，对于64kb 的page，最大row size 大约是 16000 bytes。
longblob，longtext 列必须 小于 4gb， 包含bolb，text 的 total row size 必须小于 4gb
如果 row 小于 page的一半，全部都保存到 page 中， 如果超过 page 的一半，变长列被存储到 off-page，直到 row 能放入 page 的一半。

尽管 innodb 内部支持 超过 65535 byte的 row size，但是 mysql 规定 所有列的组合后的 size 不能超过 65535.
在一些老的OS上，文件必须小于 2gb。
innodb log file 的总和 不能超过 512gb
最小的 表空间 size 比 10mb 稍小。 最大的表空间size 依赖于 page size。

InnoDB Page Size	Maximum Tablespace Size
4KB	16TB
8KB	32TB
16KB	64TB

32KB	128TB
64KB	256TB

表空间的最大值 也是 表的最大值。

一个innodb instance 支持 2^32 个 表空间，其中 少数表保留 用作 undo 和临时 table。
共享表空间 支持 2^32 个表
tablespace 文件的 路径，包括 文件名， 不能超过 window的 MAX_PATH 的限制。在 win10之前 是 260 字符，win10,1607，MAX_PATH限制被移除，但你enable 启用新的行为。

对于 并发读写事务的 限制，看 15.6.6 undo logs

==========

15.23 InnoDB Restrictions and Limitations

大于16kb 的 page 不支持 row_format=compressed。

==========
https://dev.mysql.com/doc/refman/8.0/en/replication.html
Chapter 17 Replication
- 17.1 Configuring Replication
- 17.2 Replication Implementation
- 17.3 Replication Security
- 17.4 Replication Solutions
- 17.5 Replication Notes and Tips

replication 允许 数据 从一台mysql 数据库server (称为source) 复制到 1台或多台 mysql 数据库server (称为 replicas)。
replication 默认异步。 replica 不需要 永远连着 source 来获得update。
根据配置不同，你可以 复制所有数据库，复制选中的数据库，甚至复制数据库中某些表。

mysql的复制的优点：
1. 横向扩展解决方案(scale-out solution)，将 负载 分散到多个 replica 来提升性能。在这种结构下，所有的 write 和 update 都必须在 source server 上执行。read可以发生在replica 上。这个模型可以 提升 写的性能 (因为 source 专门用来 update)，同时 可以通过 增加 replica的数量 来 提升 read 的速度。
2. 数据安全，replica 可以暂停 复制过程， 所以可以在 副本上运行 备份服务 而不会损坏相应的 数据源。
3. 分析，实时数据在 source 上生成，数据分析在 replica 上执行，这样不会 影响source

的性能
4. 远程数据分发，你可以使用 replication 来创建 远程site的 一个本地副本 来使用，而不是直接连 远程。

更多，看 17.4 replication solutions

8.0支持 不同方式的 replication。
传统的方式 是基于 从source的binary log 生成的 replicating event，在 source 和 replica 之间 同步log file 和position。
新的方式是 基于 global transaction idenfitiers(GTIDs)，这种方式是事务性的，因此 不需要处理 log文件和位置，简化了许多常见的 replication任务。 replication 使用 GTIDs 确保 source 和 replica 的 一致性，只要 所有 在 source 上 committed 的 事务 也被 应用到 replica。

使用binary log file position 进行replication，看17.1 configuration replication
使用GTIDs 进行 replication，看 17.1.3 replication with global transaction identifiers.

mysql 中的 replication 支持 不同的 同步类型。
最原始的 sync 是 单向(one-way)，异步 的 replication：一台server 扮演 source，1台或多台 server 扮演 replica。This is in contrast to the synchronous replication which is a characteristic of NDB Cluster (see Chapter 23, MySQL NDB Cluster 8.0).
8.0中，除了内置的异步replication，还支持semi-sync replication
在 semi-sync replication中，对source 进行 commit 会阻塞，直到 至少有一个 replica 回应说 已经收到并且 已经log event。
查看17.4.10 semisynchronous replication

8.0 也支持 延迟的replication，replica 故意 比source 延迟一段时间。查看 17.4.11 delayed replication。

对于 需要 synchronous replication 的场景，使用 NDB cluster (23章， MySQL NDB Cluster 8.0)

在server 间配置 replication 有很多方法，最好的方法 依赖于 你使用的 数据 和 引擎。

replication format 有2个核心类型，
基于语句的replication (statement based replication, SBR),会复制整个SQL语句。
基于row 的replication (row based replication, RBR)，只复制 被修改的row。
你也可以使用第三种，Mixed Based Replication (MBR)。

replication 被许多 option 和 variable 控制。 也可以应用 加密措施。

你可以使用 replication 来解决许多问题，包括 性能，不同数据库的备份， 作为 更大的 减少系统故障的 方案 的 一部分。 看17.4 replication solutions

==========

17.1 Configuring Replication
17.1.1 Binary Log File Position Based Replication Configuration Overview

17.1.2 Setting Up Binary Log File Position Based Replication
17.1.3 Replication with Global Transaction Identifiers
17.1.4 Changing GTID Mode on Online Servers
17.1.5 MySQL Multi-Source Replication
17.1.6 Replication and Binary Logging Options and Variables
17.1.7 Common Replication Administration Tasks

使用 binary log file position 来在2台或更多server 间 replication 的配置步骤，看 17.1.2

使用 GTID transactions 来 replication，看 17.1.3

binary log 中的 event 使用了 一些 format。它们被称为 statement based replication 或 row based replication。 第三类型，mixed format replication，自动 使用 SBR 或 RBR，来同时获得 SBR 和 RBR 的好处。

==========
https://dev.mysql.com/doc/refman/8.0/en/binlog-replication-configuration-overview.html
17.1.1 Binary Log File Position Based Replication Configuration Overview

扮演source 的 mysql 实例 将 更新和修改 作为 event 写入到 binary log。
binary log 中的 信息 根据 不同的数据库修改 以 不同的 logging format 记录。

replica 被配置，来读取 source 的 binary log 并且 在 replica 自己本次的数据库上 执行 event。

每个replica 收到 binary log 全部内容的 副本。 每个replica自己决定 binary log 中的 哪些语句 需要被执行。 除非你指定，否则 source的 binary log 中的 全部event 都会在 replica 上执行。 如果有需要，你可以 配置 replica 只 处理 应用到 指定数据库 或表 的 event。

无法 配置 source 来记录 指定的event。 (。。就是 全部记录)

每个 replica 维护了 binary log的坐标： 文件名 和 文件中的位置。
这意味着，不同的 replica 可以 执行不同的 同一个 binary log 的不同part。
由于 replica 控制了这个操作，所以 replica 的 连接 和 断开 并不会影响 source。

source 和每个 replica 都必须 通过 server_id 系统变量 配置一个 唯一的ID 。
每个 replica 必须配置 source 的 hostname， log file name, 文件的position。 这些信息可以通过 在 replica 的 mysql session 中 使用 change replication source to语句(>= 8.0.23) 或 change master to(<8.0.23) 语句 来控制。
这些信息被保存在 replica 的 connection metadata repository 中。

==========
https://dev.mysql.com/doc/refman/8.0/en/replication-howto.html
17.1.2 Setting Up Binary Log File Position Based Replication
- 17.1.2.1 Setting the Replication Source Configuration
- 17.1.2.2 Setting the Replica Configuration
- 17.1.2.3 Creating a User for Replication
- 17.1.2.4 Obtaining the Replication Source Binary Log Coordinates
- 17.1.2.5 Choosing a Method for Data Snapshots
- 17.1.2.6 Setting Up Replicas
- 17.1.2.7 Setting the Source Configuration on the Replica
- 17.1.2.8 Adding Replicas to a Replication Environment

Tips:
要部署多个mysql 实例，你可以使用 InnoDB Cluster，它可以让你通过mysql shell方便地管理一组mysql server 实例。
innodb cluster 封装了 mysql group replication，可以让你 方便地 部署 mysql 实例的 集群 来获得 高可用。
另外，innodb cluster 接口 和 mysql router 无缝衔接，可以让你 的应用 直接连到 cluster，而不需要 编写你自己的 故障处理。
对于不需要 高可用的 情况，你可以 使用 innodb replicaset。

下面是 所有配置 都需要进行的任务
1. 在source上，你必须确保 binary logging 被启用，并配置一个 独一无二的 server id。这可能需要重启server
2. 在每个 replica，你必须配置一个 唯一的server id。可能需要重启 server
3. 可选的，为你的 replica 创建一个 用户 用于 和source进行身份认证，并 读取 binary log。
4. 在 创建 data snapshot 或 启动 replication 过程 之前，在source上，你应该记录 binary log 当前的 位置。 你在配置 replica 的时候 需要这个信息，来让 replica 知道 从binary log 的哪里开始 执行 event。
5. 如果source 上已经有数据，并且 想把它 同步到 replica，你需要 创建一个 data snapshot 来 复制数据到 replica。 你所使用的存储引擎 影响了 创建snapshot 的方式。 当你使用 MyISAM时，你必须 在source 上停止处理SQL语句，来获得 read-lock，然后 获得 它的当前的binary log的文件名和位置，并 dump 数据 。 如果使用 Innodb，不需要 read-lock， 事务已经足够了。
6. 配置 replica 的 用于连接source 的信息，如 hostname, login credentials, binary log file name, position。
7. 实现适合你的系统的 对于source 和replica 的 专用于复制的 安全措施。

基础配置完后，根据场景不同：
1. 为 全新的没有数据的 source和replica 配置replication
2. 使用现有 mysql server 的 data 的 新source 的 replication
3. 在已有的 replication 环境中 新增 replica。

在管理 mysql replication server 之前， 阅读本章，并 尝试 所有语句 在：13.4.1 sql statements for controlling source servers, 13.4.2 sql statements for controlling replica servers。 也需要熟悉 17.1.6 replication and binary logging options and variables 中的 选项。

==========

17.1.2.1 Setting the Replication Source Configuration
确保 启用binary logging，并且有 唯一的server id，然后配置 source 来 使用基于binary log 的replication。

每个 replication 架构中的 server 都必须有一个 唯一的server id，通过 server_id 系统变量来指定，必须是 [1, 2^32 - 1]。 8.0开始默认是1。之前默认0。

```
SET GLOBAL server_id = 2;
```

如果之前 server id 是0， 那么需要 重启server 来初始化。 如果之前不是0 ，不需要重启。

binary logging 默认起效，即 log_bin 系统变量默认是 ON。
--log-bin 选项 告诉 server binary log文件的 基础名字。 建议你 指定这个 选项，让 binary log 不使用 默认的 base name，这样，如果host 名字改变了，不会影响 binary log file name。

如果之前 通过 --skip-log-bin 选项 禁用了 binary logging，那么需要 重启 source server。

下面2个选项也对 source 有影响：
为了 在使用innodb 的事务 时 获得最大的 持久性和一致性，你应该 使用 innodb_flush_log_at_trx_commit=1 和 sync_binlog=1，配置到 source 的 my.cnf 文件
确保 source 上 skip_networking 没有启用。

==========
https://dev.mysql.com/doc/refman/8.0/en/replication-howto-slavebaseconfig.html
17.1.2.2 Setting the Replica Configuration

每个replica 需要一个 唯一的server id。 必须>0， 如果是0，需要重启server，非0的情况下，修改server id，不需要重启server
server_id 默认 1

```
SET GLOBAL server_id = 21;
```

如果你关闭了 replica server， 你可以 编辑配置文件 来指定 server id

```
[mysqld]
server-id=21
```

binary logging 默认启用。 对于 replication 的 replica，binary logging 不是必须的。

但是 binary log 可以用于 数据备份 和 故障恢复。 replica 的binary logging 可以用于更复杂的 replication 架构，
你可以使用 链式(A->B->C) 来进行 replication，此时 A是 replica B 的 source， B 是 replica C 的source， B必须同时是 source 和 replica， 来自A 的更新 必须 log 到 B 的 binary log，然后才能 传给 C。 除了 binary logging，这种 replication 架构 要求：log_replica_updates(>=8.0.26) 或 log_slave_updates(<8.0.26) 被启用。这2个默认启用。replica update 启用后，replica 将 来自source的 并且会由replica的后台线程执行 的 update 写入到 replica 自己的binary log。

如果你需要 在 replica上 禁用 binary logging 或 replica update logging， 你可以： 指定 --skip-log-bin 和 --log-replica-updates=OFF / --log-slave-updates=OFF。 如果要 re-enable，那么需要 移除这些选项，并重启server

==========

17.1.2.3 Creating a User for Replication

每个 replica 使用 mysql user name/password 来连接source，所以 source 必须开放 一个 user 给 replica。
user name 通过 change replication source to 语句(>=8.0.23) 或 change master to 语句(<8.0.23) 的 source_user / master_user 选项 来 在 replica上 指定。 用户 必须有 replication slave 权限。 你可以为 每个replica 创建 一个account ，也可以让它们使用同一个 account。

你必须知道：user name 和 password 是 明文保存在replica 的 连接元数据仓库(mysql.slave_master_info) 中的。 因此你最好 创建一个专门的 用于 replication 的账户。

要创建新的 account，使用 create user 语句。 使用 grant 来给与权限。
下面 在source上 配置新用户 repl，这个用户 能从 example.com 域名下的 任何 host 上连接到source 进行 replication。

```
mysql> CREATE USER 'repl'@'%.example.com' IDENTIFIED BY 'password';
mysql> GRANT REPLICATION SLAVE ON *.* TO 'repl'@'%.example.com';
```

重要：
要 通过 使用caching_sha2_password 插件进行 身份认证 的 账户 来连接到 source，你必须：要么 如17.3.1中描述的 来设置一个 secure connection， 要么 启用 unencrypted连接 来 支持 使用 RSA key pair 进行 password exchange。
从8.0 开始，创建的 新user 默认 使用 caching_sha2_password。

==========
https://dev.mysql.com/doc/refman/8.0/en/replication-howto-masterstatus.html
17.1.2.4 Obtaining the Replication Source Binary Log Coordinates

要让 replica 从正确的点 开始 replication， 你需要 注意 source 的 当前的 binary log 的坐标。

警告
本过程 使用了 flush tables with read lock， 会阻塞 commit

如果你计划 将source 停机 来创建 data snapshot， 你可以 不执行这个过程，而是 将 binary log index file 的副本和 数据快照一起存储。在这种情况下，source 在 重启时 创建新的 binary log file。

要获得 source binary log 坐标，步骤如下：
1. 在source上启动一个 session，通过下列语句 flush 表和 块写入语句：
   mysql> FLUSH TABLES WITH READ LOCK;
   保持client 以确保 read lock 有效，如果退出客户端，lock被释放。
   。。。这个锁 在 17.1.2.6 中才释放， 需要 看完全部。

2. 在source 上的另一个 session，执行 show master status 语句 来决定 当前 binary log file name 和 position
   mysql > SHOW MASTER STATUS;
   ```
   +------------------+----------+--------------+------------------+
   | File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
   +------------------+----------+--------------+------------------+
   | mysql-bin.000003 | 73       | test         | manual,mysql     |
   +------------------+----------+--------------+------------------+
   ```
   file列 展示了 log file 的名字，position列展示了 文件中的位置。
   如果 source 是在 binary logging 禁用的情况下 允许，那么 log file name 和 position 会是空的。 这种情况下，你需要 使用 空string('') 作为 log file name， 4 作为 position。

现在你已经有 用于 配置replica 的数据。
下一步取决于 source 是否存在数据
1. 如果在 开始 replication 之前 存在数据 需要 同步到 replica，让客户端继续执行，以便继续lock住。看17.1.2.5
2. 如果你配置一个 新的source 和 replica，你可以 退出 第一个session 以释放lock。看17.1.2.6.1

==========
17.1.2.5 Choosing a Method for Data Snapshots
如果 source 上已经有数据需要同步到 每个 replica。有不同的方法 来 从 source 数据库 dump 数据。
17.1.2.5.1 Creating a Data Snapshot Using mysqldump

跳
17.1.2.5.2 Creating a Data Snapshot Using Raw Data Files
跳

==========

17.1.2.6 Setting Up Replicas

下面描述 如何配置 replica，在你执行前，确保你已经：

1. 已经 对 source 进行了 必要的配置，看17.1.2.1 setting the replication source configuration
2. 获得 source status 信息，或 source 为了 data snapshot 而shutdown 期间 制作的 binary log index 文件。看17.1.2.4
3. 在source，解锁
   ```
   mysql> UNLOCK TABLES;
   ```
4. 在replica，编辑 mysql config，看 17.1.2.2

下一步取决于 source 上是否 已经有 需要同步的数据 需要 导入到 replica
如果不需要导入，看17.1.2.6.1
如果需要导入，看17.1.2.6.2

17.1.2.6.1 Setting Up Replication with New Source and Replicas

17.1.2.6.2 Setting Up Replication with Existing Data

==========

17.1.2.7 Setting the Source Configuration on the Replica

```
mysql> CHANGE MASTER TO
    ->     MASTER_HOST='source_host_name',
    ->     MASTER_USER='replication_user_name',
    ->     MASTER_PASSWORD='replication_password',
    ->     MASTER_LOG_FILE='recorded_log_file_name',
    ->     MASTER_LOG_POS=recorded_log_position;

Or from MySQL 8.0.23:
mysql> CHANGE REPLICATION SOURCE TO
    ->     SOURCE_HOST='source_host_name',
    ->     SOURCE_USER='replication_user_name',
    ->     SOURCE_PASSWORD='replication_password',
    ->     SOURCE_LOG_FILE='recorded_log_file_name',
    ->     SOURCE_LOG_POS=recorded_log_position;
```

==========
17.1.2.8 Adding Replicas to a Replication Environment

==========
https://dev.mysql.com/doc/refman/8.0/en/replication-gtids.html
17.1.3 Replication with Global Transaction Identifiers
- 17.1.3.1 GTID Format and Storage
- 17.1.3.2 GTID Life Cycle
- 17.1.3.3 GTID Auto-Positioning
- 17.1.3.4 Setting Up Replication Using GTIDs
- 17.1.3.5 Using GTIDs for Failover and Scaleout
- 17.1.3.6 Replication From a Source Without GTIDs to a Replica With GTIDs
- 17.1.3.7 Restrictions on Replication with GTIDs
- 17.1.3.8 Stored Function Examples to Manipulate GTIDs

使用 global transaction identifiers(GTIDs) 进行 基于事务的 replication。
当使用 GTIDs 时，在source上 commit 并被应用到 每个replica 的 每个事务 都可以被 识别和 跟踪。 这意味着 使用GTIDs 不需要记录 log file name 和position。意味 基于GTID的replication 是 基于事务的，很容易就 确定 source 和 replica 是否 一致，只要 source 上所有 commit 的事务 在 replica 上也 commit了，那么就是一致的。
你可以将 基于语句 或 基于row 的 replication 和 GTID 一起使用。为了最好的效果，建议你使用 row-based。

==========

17.1.3.1 GTID Format and Storage

mysql.gtid_executed

会记录下执行的 GTID，如果已经执行了就自动跳过， 如果 一个GTID事务在执行，又开启一个相同GTID的事务，后者会 block 直到第一个 commit 或rollback，commit的话第二个就跳过，rollback的话，第二个就继续执行。

GTID = source_id:transaction_id

source_id 通常是 server_uuid。 transaction_id 是 sequence number，由 事务在source 上 commit 的次序 决定。比如，第一个commit的事务的 transaction_id 是1，第十个commit 的事务 的 transaction_id 是10。 transaction_id 不可能是0。

3E11FA47-71CA-11E1-9E33-C80AA9429562:23

sequence number 是一个 有符号64bit integer 的 >=1 部分。 如果 用完，会执行 binlog_error_action 中定义的 操作。 从8.0.23开始，如果 接近这个 限制，会有 warning。

事务的 GTID 展现在 mysqlbinglog 的输出中，值保存在 gtid_next 系统变量 (@@GLOBAL.gtid_next)。

GTID Sets
3E11FA47-71CA-11E1-9E33-C80AA9429562:1-5
3E11FA47-71CA-11E1-9E33-C80AA9429562:1-3:11:47-49
2174B383-5441-11E8-B90A-C80AA9429562:1-3, 24DA167-0C0C-11E8-8442-00059A3C7B00:1-19

mysql.gtid_executed Table
使用 reset master 语句时，mysql.gtid_executed 被清空。

只有 gtid_mode 是 ON 或 ON_PERMISSIE 时， 才会 保存。
如果 binary logging 被禁用，或

。。跳

==========

https://dev.mysql.com/doc/refman/8.0/en/replication-multi-source.html
17.1.5 MySQL Multi-Source Replication
- 17.1.5.1 Configuring Multi-Source Replication
- 17.1.5.2 Provisioning a Multi-Source Replica for GTID-Based Replication
- 17.1.5.3 Adding GTID-Based Sources to a Multi-Source Replica
- 17.1.5.4 Adding Binary Log Based Replication Sources to a Multi-Source Replica
- 17.1.5.5 Starting Multi-Source Replicas
- 17.1.5.6 Stopping Multi-Source Replicas
- 17.1.5.7 Resetting Multi-Source Replicas
- 17.1.5.8 Monitoring Multi-Source Replication

==========

==========

==========

https://dev.mysql.com/doc/refman/8.0/en/replication-solutions.html

17.4 Replication Solutions

- 17.4.1 Using Replication for Backups
- 17.4.2 Handling an Unexpected Halt of a Replica
- 17.4.3 Monitoring Row-based Replication
- 17.4.4 Using Replication with Different Source and Replica Storage Engines
- 17.4.5 Using Replication for Scale-Out
- 17.4.6 Replicating Different Databases to Different Replicas
- 17.4.7 Improving Replication Performance
- 17.4.8 Switching Sources During Failover
- 17.4.9 Switching Sources and Replicas with Asynchronous Connection Failover
- 17.4.10 Semisynchronous Replication
- 17.4.11 Delayed Replication

==========

https://dev.mysql.com/doc/refman/8.0/en/replication-solutions-backups.html

17.4.1 Using Replication for Backups

- 17.4.1.1 Backing Up a Replica Using mysqldump
- 17.4.1.2 Backing Up Raw Data from a Replica
- 17.4.1.3 Backing Up a Source or Replica by Making It Read Only

==========

https://dev.mysql.com/doc/refman/8.0/en/group-replication.html

Chapter 18 Group Replication

18.1 Group Replication Background
18.2 Getting Started
18.3 Requirements and Limitations
18.4 Monitoring Group Replication

18.5 Group Replication Operations
18.6 Group Replication Security
18.7 Group Replication Performance and Troubleshooting
18.8 Upgrading Group Replication
18.9 Group Replication System Variables
18.10 Frequently Asked Questions

==========

https://dev.mysql.com/doc/refman/8.0/en/mysql-innodb-cluster-introduction.html
Chapter 21 InnoDB Cluster

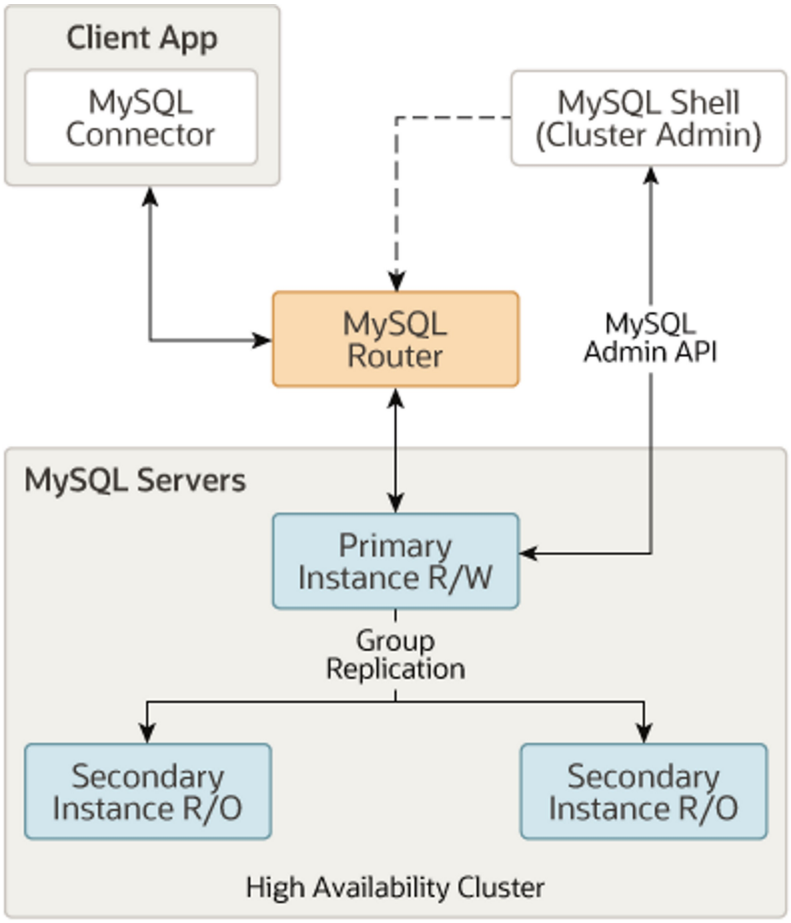
mysql innodb cluster，整合了 mysql 的技术 来让你 部署和管理 mysql的完整集成的高可用解决方案。
本章只是一个 概述，具体：https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-innodb-cluster.html

重要：
innodb cluster 不支持 mysql ndb cluster
。。NDB 也称为NDB CLUSTER, 是另一种存储引擎,但是它主要存储数据在内存中,并且独立于MySQL Server实例。它是MySQL Cluster使用的存储引擎。 NDB代表“网络数据库”。

一个innodb cluster 至少包含 3台 mysql server 实例， 提供了 高可用 和 伸缩性。
innodb cluster 使用了下列 mysql 技术

1. mysql shell，mysql的 一个 高级client 和代码编辑器
2. mysql server 和 group replication，让 一组mysql 实例 提供 高可用。innodb cluster 提供了一种替代的，易于使用的编程方式 来处理 group replication。
3. mysql router，轻量级中间件， 为 你的应用 和 innodb cluster 提供 透明的路由。

下图展示了这些技术的组合



。。高可用 和扩展。。 不是 高性能。 不过可以 读写分离。 主写，从读。
。。而且 17.4.6 可以将不同数据库 复制到不同的 replica，不知道 不同表 可不可以。 那就 分库分表了， 但也不算分库分表。。 但是 对于 写少 读多的，这种真的很适合啊。

基于 mysql group replication，提供了 自动membership 管理，容错，自动故障转移 等。
innodb cluster 通常 以 单主模式(single-primary mode) 运行，一台主实例(read-write)，多台从实例(secondary instance)(只读)。
高级用户可以 使用 multi-primary mode 。(https://dev.mysql.com/doc/refman/8.0/en/group-replication-multi-primary-mode.html) 。看起来 这个 高级用户 并不是 付费的意思，不知道哪里 advanced 了。
在这个模式下，所有实例都是 primary。
你可以在 innodb cluster online 的时候 修改 cluster 的拓扑，来获得 最高的可用性。

你通过 mysql shell 的 AdminAPI 来 和 innodb cluster 一起工作。
AdminAPI 可以使用 JavaScript 和 python，非常适合 将mysql的部署 进行脚本化 以自动部署 来获得 高可用和伸缩性。
通过使用 mysql shell 的 adminapi，你可以 避免 手工配置 许多 实例。 adminapi提供了高效现代的接口 来 设置 mysql instance，可以让你 准备，管理，监控 你的 部署

innodb cluster 支持 mysql Clone，可以让你 简单地 准备你的实例。

==========

Chapter 22 InnoDB ReplicaSet

==========

Chapter 23 MySQL NDB Cluster 8.0

23.1 General Information
23.2 NDB Cluster Overview
23.3 NDB Cluster Installation
23.4 Configuration of NDB Cluster
23.5 NDB Cluster Programs
23.6 Management of NDB Cluster
23.7 NDB Cluster Replication
23.8 NDB Cluster Release Notes

MySQL NDB Cluster uses the MySQL server with the NDB storage engine.

==========

https://dev.mysql.com/doc/refman/8.0/en/partitioning.html
Chapter 24 Partitioning

24.1 Overview of Partitioning in MySQL
24.2 Partitioning Types
24.3 Partition Management
24.4 Partition Pruning
24.5 Partition Selection
24.6 Restrictions and Limitations on Partitioning

讨论 user-defined partitioning.

8.0中， innodb 和 NDB 存储引擎 支持 partitioning。其他存储引擎不行。

。。这个要自己编译？

==========

24.1 Overview of Partitioning in MySQL

SQL标准 并没有提供 数据存储的物理方面的 指导。
SQL语言 想要 独立于 schema，tables，row，column 的 数据结构 和载体。
但是，很多先进的DBMS 能够 为 指定的数据 决定物理位置。
innodb 支持 tablespace 的概念。
mysql 在引入 partitioning 之前 就可以 将 不同的数据库 存放到 不同的 物理目录下。

partitioning 使得这个概念进一步发展，通过 允许你 根据(大量需要你配置的)规则 来将 表的 部分 分发到 文件系统中。
效果是： 表的不同部分 作为单独的表 被存储到 不同位置。
用户定义的 如何切分数据 的 规则 被称为 partitioning function，在mysql 中 它可以是 模数，与一组范围 或 值列表 的简单匹配，内部哈希函数 或 线性哈希函数。 该函数根据用户指定的 分区类型 进行选择，并将 用户提供的 表达式的值 作为其参数， 表达式可以是 列值，作用于一个或多个列值的函数，也可以是 一组一个或更多列值，具体取决于 所使用的 分区类型。

在 range，list和 [linear] hash partitioning中，partitioning列的值 被传递到 partitioning function，这个function会返回一个 integer 值 ，表示 这个record 应该存储的 分区。
这个function 必须是 non-constant，non-random 的。 它可能不包含任何查询，但是可能 使用 一个 SQL表达式，只要 那个SQL表达式 返回 一个 null 或 一个 integer。

对于 [linear] key，range columns，list columns 分区， partitioning 表达式 包含 1 或更多 列的 list。

对于 [linear] key 分区，partitioning function 由 mysql 支持。

这就是 horizontal partitioning (水平分区)，即， 一个table 的不同 row 可能存储于 不同的 物理分区中。
8.0 不支持 vertical partitioning (垂直分区)，一个table 的 不同 column 存储于 不同的物理分区中。 目前还没有计划 在 mysql 中引入 垂直分区。

为了创建分区后的表， 你必须使用 支持它的 存储引擎。
8.0中，一个表的 所有 分区表 都必须使用 相同的 存储引擎。 不同的表 可以使用 不同的存储引擎。

8.0中，只有 Innodb 和 NDB 支持 partitioning。 其他的 不支持 (如 MyISAM，MERGE，CSV，Federated)

NDB 可以使用 按key 或 linear key 的分区， 其他的 用户定义的 partitioning 不支持。
使用 用户定义的 partitioning 的 NDB 必须有一个 显式 主键， 并且 用于 partitioning expression 的 所有列 都必须是 主键的 一部分。
但是，用于 创建 或修改 user-partitioned NDB table 的 create table 或 alter table 语句的 partition by key 或 partition by linear key 子句 中 没有 column， 那么 表 不需要 有一个 显式 主键。

创建 partitioned table 时，默认存储引擎 和 创建其他表 一样。 要使用 [storage] engine 选项 来指定存储引擎。
你必须记住， [storage] engine 必须 在 create table 的 所有其他 partitioning option

之前。

```
CREATE TABLE ti (id INT, amount DECIMAL(7,2), tr_date DATE)
    ENGINE=INNODB
    PARTITION BY HASH( MONTH(tr_date) )
    PARTITIONS 6;
```

每个 partition 子句 都可以 包含 一个 [sotrage] engine 选项，但是 在 8.0 中，这个 并没有 效果。

之后的例子中， default_storage_engine 是 InnoDB。

重要：

partitioning 会应用到 表的 所有数据 和 index， 你不可能 只 对 数据 partition，不对 index partition

使用 create table 的 partition子句的 data directory 和 index directory 选项 可以 设置 每个分区的 data 和index。

InnoDB 表的 partition 和 subpartition 只支持 data directory 选项。从8.0.21 开始，在 data directory 中指定的 目录 必须被 innodb 知道。

用于 table 的 partition 表达式 的 所有 列 必须是 这个表的 每个unique key (包括所有主键) 的 一部分。这意味着，下面的表， 不能被 partition：

```
CREATE TABLE tnp (
    id INT NOT NULL AUTO_INCREMENT,
    ref BIGINT NOT NULL,
    name VARCHAR(255),
    PRIMARY KEY pk (id),
    UNIQUE KEY uk (name)
);
```

因为 key： pk 和 uk， 没有 common 列，所以 没有列 可以用于 partition 表达式。可能的解决方案是 将 name 列 加入到 主键中，将 id 列 增加到 uk 中， 或者 移除 unique key。

另外，max_rows 和 min_rows 选项 可以用来决定 每个partition 中 存储的 row 的 最大 和 最小 数量。

max_rows 选项 还可以用于 创建 具有额外分区的 NDB 集群表，从而允许存储更多的哈希列。查看 DataMemory data node 的文档

partitioning 的一些优势

1. 分区 可以让 一张table 存储更多的 数据，相比 单个disk 或 文件系统分区
2. 如果数据没有用了，那么可以 将 包含这些无用数据的 partition 直接 drop 掉 来 来简单地 从 partitioned table 中 移除这些数据。 另一方面，通过 添加 一个或多个 新分区 来专门存储 数据，可以大大简化 添加新数据的 过程。
3. 如果 给定where 子句 的 数据 可能 被存储于 1个或多个 partition 中，那么 其他的 partition 就不会被 搜索， 这可以 大大优化 query。 因为 分区 可以在 创建分区表后 进行 修改，所以 可以重新组织 数据 以 增强 ，在之前 设计方案时 没有考虑到的 频繁的查询。 这种排除 不匹配分区 的能力 通常被称为 分区修剪(partitioning pruning)

另外，mysql 支持 query 时 指定 分区。 select * from table1 partition (p0,p1) where c < 5。 只在 分区 p0 和 p1 中搜索 匹配where 的row。可以大大提高query 速度。 指定partition 也可以用于 数据修改语句： delete,insert,replace,update,load data,load xml。

==========

https://dev.mysql.com/doc/refman/8.0/en/partitioning-types.html

24.2 Partitioning Types

- 24.2.1 RANGE Partitioning
- 24.2.2 LIST Partitioning
- 24.2.3 COLUMNS Partitioning
- 24.2.4 HASH Partitioning
- 24.2.5 KEY Partitioning
- 24.2.6 Subpartitioning
- 24.2.7 How MySQL Partitioning Handles NULL

介绍8.0 可用的partitioning 类型：

1. range partitioning, 根据 给定range 内的列值 来将 row 分区。
2. list partitioning，类似 range，除了 这里是 用 列值 和 离散值的集合 进行比较。
3. hash，基于 用户定义的 (基于将被插入的row 上列 的) 表达式 的值 来选择 分区。
   有一个扩展， linear hash
4. key，类似hash，除了 只能eval 一个或多个列，mysql 提供了它自己的 hashing 函数。这些列 可以包含 非整数值， mysql的hashing 函数 保证 返回 一个整数，不管列的类型。
   有一个扩展，linear key

partitioning 的一个很常用的 用例是 按日期 分隔 数据。
一些数据库系统 支持 显式的 date partitioning， mysql 8.0 没有实现。 但是 对于mysql 创建一个 基于 date,time或datetime列 或 基于它们的表达式 的 partitioning，不难。

使用 key，linear key 分区，你可以使用 data, time or datetime列 作为分区列， 不需要修改 列值:

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL,
    lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL,
    email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY KEY(joined)
PARTITIONS 6;
```

使用 range,list 分区，你可以使用 date 或 datetime 列 作为 range columns 和 list columns 的值。

其他 需要分区表达式(产生null 或integer)的 分区类型。如果你想 通过 range ,list,hash,linear hash 来 进行 基于时间分区， 你可以 使用一个 对 date,time or datetime 列操作的 函数：

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL,
    lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL,
    email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY RANGE( YEAR(joined) ) (
    PARTITION p0 VALUES LESS THAN (1960),
    PARTITION p1 VALUES LESS THAN (1970),
    PARTITION p2 VALUES LESS THAN (1980),
    PARTITION p3 VALUES LESS THAN (1990),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

mysql 的分区，在使用 to_days(),year(),to_seconds() 函数时 会进行优化， 当然， 你可以使用 其他的 返回 integer 或 null 的 日期和时间函数，比如 weekday(),dayofyear(),month()。

有一点要记住：

不管你使用了 什么 分区类型，分区 在创建时 总是 自动按顺序编号，从0开始。

如果 table 有4个分区，那么是0,1,2,3。

。。它应该会 mod 下吧。

partition 的名字 是大小写不敏感的，下面的 sql 会报错

```
mysql> CREATE TABLE t2 (val INT)
    -> PARTITION BY LIST(val)(
    ->     PARTITION mypart VALUES IN (1,3,5),
    ->     PARTITION MyPart VALUES IN (2,4,6)
    -> );
ERROR 1488 (HY000): Duplicate partition name mypart
```

==========

range

list

range columns
list columns
hash
linear hash
key linear key

==========

https://dev.mysql.com/doc/refman/8.0/en/partitioning-subpartitions.html
24.2.6 Subpartitioning

也称为 composite partitioning，复合分区。

```
CREATE TABLE ts (id INT, purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) )
    SUBPARTITION BY HASH( TO_DAYS(purchased) )
    SUBPARTITIONS 2 (
        PARTITION p0 VALUES LESS THAN (1990),
        PARTITION p1 VALUES LESS THAN (2000),
        PARTITION p2 VALUES LESS THAN MAXVALUE
    );
```

。。ABBA

表有 3个 range 分区，每个分区 进一步 分为 2个 子分区， 所以 整个表 被分为 3*2=6 个分区。 根据 partition by range 子句，前2个 分区 只保存了 purchased 列 小于 1990 的数据。

可以对 range，list 分区的 表 进行 子分区，子分区 可以是 hash 或 key 分区。

。。跳

==========

24.2.7 How MySQL Partitioning Handles NULL

==========

24.3 Partition Management
    24.3.1 Management of RANGE and LIST Partitions
    24.3.2 Management of HASH and KEY Partitions
    24.3.3 Exchanging Partitions and Subpartitions with Tables
    24.3.4 Maintenance of Partitions
    24.3.5 Obtaining Information About Partitions

可以 使用 alter table 的分区扩展 来 add, drop, redefine, merge, split 分区。

==========

24.4 Partition Pruning
分区剪枝

```
CREATE TABLE t1 (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    region_code TINYINT UNSIGNED NOT NULL,
    dob DATE NOT NULL
)
PARTITION BY RANGE( region_code ) (
    PARTITION p0 VALUES LESS THAN (64),
    PARTITION p1 VALUES LESS THAN (128),
    PARTITION p2 VALUES LESS THAN (192),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

```
SELECT fname, lname, region_code, dob
    FROM t1
    WHERE region_code > 125 AND region_code < 130;
```

当 where 子句可以 变成 下面的 2种方式 之一， 优化器 可以 分区剪枝。
    partition_column = constant
    partition_column IN (constant1, constant2, ..., constantN)

==========

24.5 Partition Selection

```
mysql> SELECT * FROM employees PARTITION (p1);
```

```
mysql> SELECT * FROM employees PARTITION (p0, p2)
    ->     WHERE lname LIKE 'S%';

mysql> SELECT store_id, COUNT(department_id) AS c
    ->     FROM employees PARTITION (p1,p2,p3)
    ->     GROUP BY store_id HAVING c > 4;


mysql> SELECT id, CONCAT(fname, ' ', lname) AS name
    ->     FROM employees_sub PARTITION (p2sp1);

mysql> INSERT INTO employees_copy
    ->     SELECT * FROM employees PARTITION (p2);


mysql> SELECT
    ->     e.id AS 'Employee ID', CONCAT(e.fname, ' ', e.lname) AS Name,
    ->     s.city AS City, d.name AS department
    -> FROM employees AS e
    ->     JOIN stores PARTITION (p1) AS s ON e.store_id=s.id
    ->     JOIN departments PARTITION (p0) AS d ON e.department_id=d.id
    -> ORDER BY e.lname;

mysql> UPDATE employees PARTITION (p2)
    ->     SET store_id = 2 WHERE fname = 'Jill';


mysql> INSERT INTO employees PARTITION (p2) VALUES (20, 'Jan', 'Jones', 1, 3);
ERROR 1729 (HY000): Found a row not matching the given partition set
mysql> INSERT INTO employees PARTITION (p3) VALUES (20, 'Jan', 'Jones', 1, 3);
Query OK, 1 row affected (0.07 sec)

mysql> REPLACE INTO employees PARTITION (p0) VALUES (20, 'Jan', 'Jones', 3, 2);
ERROR 1729 (HY000): Found a row not matching the given partition set

mysql> REPLACE INTO employees PARTITION (p3) VALUES (20, 'Jan', 'Jones', 3, 2);
Query OK, 2 rows affected (0.09 sec)

mysql> ALTER TABLE employees
    ->     REORGANIZE PARTITION p3 INTO (
    ->         PARTITION p3 VALUES LESS THAN (20),
    ->         PARTITION p4 VALUES LESS THAN (25),
    ->         PARTITION p5 VALUES LESS THAN MAXVALUE
    ->     );
```

==========

https://dev.mysql.com/doc/refman/8.0/en/partitioning-limitations.html
24.6 Restrictions and Limitations on Partitioning
    24.6.1 Partitioning Keys, Primary Keys, and Unique Keys
    24.6.2 Partitioning Limitations Relating to Storage Engines
    24.6.3 Partitioning Limitations Relating to Functions

==========

Chapter 25 Stored Objects

==========

==========

==========

==========

==========

==========

==========

==========

==========

https://dev.mysql.com/doc/refman/8.0/en/glossary.html

MySQL 术语

==========

==========

==========